CPS 720 Artificial Intelligence Topics with Agents

Fall 2001

Original notes by D. Grimshaw

This course focuses on software agents, particularly mobile agents. The programming language used is Java. Several agent API's are discussed. These include Aglets, originally from IBM, now Open Source, the Java Agent Development Environment (JADE) from the University of Parma, and Ascape, from the Brookings Institute in Washington DC. Communication languages such as the Semantic Language (SL) and XML will also be discussed.

Disclaimer

General Course Information

- For last year's course (Fall 2000)
- Course Management Form (Fall 2001)
- **Course Resources and References**
- Assignments
- **Exam ReadMe**

Course Topics

- Introduction: what is an agent?
  - Agents: Natural and Artificial
  - Ferber's Discussion
  - Lange and Oshima
  - Nikolic
- Situated Agents
  - Agent rationality
  - Agent autonomy
  - An Agent Classification Scheme
  - A Basic Reactive Agent Example
  - A Reactive Agent with state
  - Agent environments

- Mobile Communicative Agents
  - The CERN CSC99 Agent Course
    - Lecture 1
      - Notes on Lecture 1
    - Lecture 2
      - Notes on Lecture 2
    - Lecture 3
  - The Agent and its environment
  - Seven Advantages of mobility
  - Mobile agents vs mobile objects
  - Agents and network computing paradigms
  - The question of location transparency in distributed systems
- Aglets
  - What are Aglets?
  - Getting started with Aglets
  - The Aglet Model
  - The Aglet Package
    - Aglet Mobility
    - Inter-Aglet Communication
    - Cloning
    - Aglet Design Patterns
      - Survey of Patterns
      - The Master-Slave Pattern
      - The Sequential Itinerary Pattern
      - Combining the Master-Slave and Itinerary Patterns
      - The SeqItinerary and SlaveItinerary patterns of the Aglets API
  - Aglets as Agents
- Agent Communication
  - Multi-Agent Systems: example
  - The Semantic Web
  - Communicative Acts
  - Agent Communication Languages
  - ContentLanguages
  - Ontology

- JADE
  - Getting started with JADE
  - The FIPA Agent Model and JADE
  - The JADE 2.4 API and other docs (local)
  - Ontologies and JADE
  - Programming with JADE
- XML
  - Survey
    - Introduction
    - Using XML
    - Document Type Definitions
    - XML and Inter-Agent Communication
  - XML Interpretation using the DOM
  - Using the SAX API
  - XSL and XSLT
- Agent Negotiations
  - Introduction
  - The Problem of Deception
  - Zero Sum Games
  - Cooperative Games
- Agents and Simulation
  - Ascape
    - Ascape Notes and Tutorials

Disclaimer

The CPS720 notes are lecture notes. They have not been peer reviewed or published in book form. They are meant to be helpful but are not necessarily complete or completely accurate. When in doubt, consult the relevant textbooks or references.

The author is not responsible for any damages, direct or indirect, caused by the use of these notes.

Course Information from Fall 2000

- Course Management Form
- Assignments
- Exam Readme, Fall 2000

CPS 720 Course Management Form

Fall 2001

Instructor: David Grimshaw

Office: Kerr Hall, QE 327A

Phone: (416) 979-5000 #6973

e-mail: dgrimsha@scs.ryerson.ca

WWW: http://www.ryerson.ca/~dgrimsha

Office Hours: See Teaching Timetable

Textbook: Danny Lange & Mitsuru Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998, ISBN 0-201-32582-9 (Not compulsory)

Evaluations:

Type	Weight (%)	Due Dates
Assignment 1	10	Fri. Sept. 28
Assignment 2	20	Fri. Oct. 19
Assignment 3	20	Fri. Nov. 30
Exam	50	Thu. Dec. 13, 9-11

Notes:

1. Students obtaining less than 40% of the total marks available from the exam will automatically fail, regardless of their marks on the assignments.
2. Students obtaining tan exam mark between 40% and 50% will have their final mark adjusted at the discretion of the instructor. This adjustment will probably result in a lowering of the student's nominal mark, but will unlikely result in a failing grade.
3. The exam will be either all multiple choice, or multiple choice with a few short answer or fill in the blank questions. A programming question is possible.

CPS 720 Assignments, Fall 2001

Java Code Conventions

Assignment 1

Assignment 2

Assignment 3

Student Port Assignments

Marks

As of Dec. 11, 2001

.

CPS 720 Assignment 1 A Simple Aglet

Due Friday, September 28, 2001, Midnight. 10 Marks

Please do this assignment individually.

In this assignment you develop an Aglet which carries some text to a remote host and writes it into a file there. Of course the Aglet must have writer permission on the remote host.

The text should be your Aglet's own Java source code. On creation, your aglet should read its source code into an aglet member (as a String or a Vector of Strings, perhaps). The data structure must be serializable. On arrival, the aglet writes the text to a file in the directory I:\coursesf01\cps720\assignment1 at atp://proton.scs.ryerson.ca:4434.

Note Sept. 12: Use 141.117.14.123. proton.scs.ryerson.ca is not working at present.)

Note Sept 19: Proton now seems to be ok. The address proton.scs.ryerson.ca is now valid.

The filename must be unique, so use the logon name, e.g., dgrimsha.java.

SCS contain both UNIX and Windows machines. This causes a potential portability problem. In UNIX the file separator is '/' whereas for MS Windows it is '\'. (At the moment, proton is a Windows NT machine.) Since your Aglet may not know in advance what kind of machine it lands on, you need to build in flexibility. Fortunately, Java provides the System.getproperty() method which you can use to get the value of the file.separator property for an OS. Also don't forget that '\' is an escape character in Java, as in C.

When creae your Aglet in your "Tahiti" Aglet server using the Create button, your Aglet should be created, and then automatically dispatched to proton. Then You can get your Aglet back too. (Retract button) In fact you need to retract your Aglet in order to see if it successfully wrote the file. You will need some extra coding on your Aglet to do this.

If the Aglet fails in its write operation on the remote host (proton) it is usually because of a security exception (not an IOException as you might expect). You can set up a String field in the Aglet and have it contain a failure or a success message depending on whether this exception is thrown or not.

The Tahiti server has a Dialog button which you can use to send the message string "dialog" to resident Aglets. Use this feature with the handleMessage() method to have the retracted Aglet display the success/failure message on stdout (or on the Tahiti window).

You should develop this assignment either on jupiter at school, or on your own PC at home. From home the Aglet should dispatch itself through your ISP without problems. The retract button should also work. (Trying to automatically have your aglet return to you, boomerang fashion) probably will not work with a home connection although it would if you stay within the SCS domain. Unless you have your own fully qualified domain name.)

CPS 720 Assignment 2 Moible Agent Survey

Due Friday, Oct. 19, 2001. 15 marks.

This is a group assignment. Groups of up to 4 are allowed. Working alone is not recommended.

Overview

This assignment has some similarity with the lab done at the CERN CSC99 course. It simulates a survey. The survey consists of a simgle statement to which respondants answer on a scale of 1 to 10. 10 means completely agree; 1 means completely disagree; 5 or 6 means neutral, or no strong opinion.

For example, how would your rate the statement "*Celine Dion is the best female vocalist today"*.

You can imagine each Aglet server (tahiti) represents a diffrent region or country where the survey takes place. The idea is to have a mobile agent visit each of these servers in turn, combine the results, and return to its origin. The total result is to be displayed as a histogram.

Specifics.

Packages

This assignment involves quite a few files so packaging is needed. You should have two packages, cps720.assignment2m and cps720.assignment2.util.

All your code except two utilites should go in cps720.assignment2. A utility class needed to convert back and forth between "csv String" histogram representaion, and array representation of histograms should be put in cps720.assignment2.util. (See below for more on these representations.)

You will also be using a graphics package to draw the histogram. This package is called PtPlot and comes from the Ptolemy project at the University of California, Berkeley. This system is in jar file called plot.jar. If you want to copy it to your system, make sure you put it in the aglets/lib directory (where the aglets jar file is) otherwise the tahiti server cannot find it.

To compile with PtPlot, you must `import ptolemy.plot.*;` into your code.

The data

Of course, we cannot conduct real surveys, so we have to fake it. You are supplied with a simple Java program, cps720.assignment2.CreateSurvey. This allows you to "guide" the results. You answer the question yourself and then the program uses a Gaussian distribution random number generator to generate 'responses" with your answer as mean. You can adjust various parameters.

The program generates a binary file containint ints. You need to look at the source code to see how to write your own conde to read the data back in.

Architecture

You need to write 3 agents, two stationary and one mobile.

The service agent

At each server there is a stationary agent which can read the survey data and pass it in appropriate format as a message in response to a message sent to it by a visiting mobile agent. (The mobile agent is not to have direct access to the data file.)

This stationary service agent responds to the messsage, **"histogram"**. **Every system MUST use this string** since you want your service agent to respond to the visiting agents written by others.

When the service agent is created, it should read in the survey data file and create a histogram representation. This datastructure is just a string containing comma separated values (csv). For example you might wind up with "21,34,56,122,132,136,119,49,28,17". The numbers here represnt survey counts. So, in the example, 21 people strongy disagreed with the statement (rating = 1), and 17 people strongly agreed with the statement (rating =10).

This representation is used in the Aglet Message method sendReply(). You might find it more natural to store the histogram results in an array of ints, due to a bug (?) in the Aglet API sendReply(Object o) does not seem to work for user defined objects. The String type is safe, however. **The use of this csv String representation is compulsory for any histogram representations to be used in messages**.

Converting from array to csv representation

The csv String representation is needed for Aglet messages, but of course it is not much use for manipulation integers. To update the histogram data you need to add ints. So you also need to keep an array of 10 int elements.

You should write a class with two static methods with signatures such as,

- public static String arrayToCsv(int [] array)
- public static int [] csvToArray(String csv)

To write thse you will find the classes String, String Buffer (with method append), and StringTokenizer (in package java.util) useful.

Note that the class SurveyHistogram used to display the final result expects these conversion methods to have exactly the signatures shown above. Also these methods must be in a class called Convert in a package cps720.assignment2.util.

The mobile agent

The mobile collector agent also must use the csv String representation. It starts with "0,0,0,0,0,0,0,0,0,0" and updates the values at each stop on its jouney. It will make good use of your converter class!

You can write the code to control the mobile agent's itinerary from scratch, or you can use the SlaveItinerary and Task classes form the package com.ibm.agletx.util (note the 'x').

This agent understands one message, "getHistogram". (You could use another string here. This message is only used to communicate between your mobile agent, and your master agent.)

The Master Agent

This agent is a static agent. It is responsible for creating the mobile agent (which starts itself on its itinerary).

The master agent responds to the Aglet "dialog" message sent by the Tahiti sever when the Dialog button is clicked. In response to this message, the master agent sends the "getHistogram" to the mobile agent. (Of course you don't send the "dialog" message until the travelling aglet returns, or is retracted.)

In response to the "getHistogram" message the returned mobile agent sends the csv String representaion back to the Master agent. In response, the master agent displays the final histogram using PtPlot.

To avoid having to learn too much about the PtPlot API you can use SurveyHistogram.java.

How to submit this assignment

All your class and java files should be packaged in one Java jar file. Include also a readme file with group names and brief descriptions of what your files do. Be sure to follow the package structure described above.

Send your jar file as an attachment to david.grimshaw@sympatico.ca, NOT a Ryerson addresss. Include group names in the main message text.

CPS 720 Assignment 3 A Supply and Demand Simulation using JADE

Due: Friday, November 30, 2001, 20 Marks

This is a group assignment. Maximum group size is 4.

Introduction

In this assignment you are to create a simulation of supply and demand from basic economics. The system consists of one producer of some product, and n consumers of the product. The producer can set a price for its product and tell all the consumers the price. Consumers each have their own demand schedule which determines how much of the product they will buy at the given price.

The consumers tell the producer how much of the product they want. The producer then calculates its profit based on the total quantity sold, the unit price and the unit cost of production. Constant returns to scale are assumed. That is, the producer can produce any amount of its product at the same unit cost.

If the producer does not like the profit at one price, it can try again with a different price. Of course a loss might also be possible.

Economics

Profit is just the difference between revenue and costs. So, if the unit price of the product is p, and the unit cost is c, and the amount sold is q, then

profit = (p - c) * q

A demand schedule is a plot of quantity demanded, q, vs. price, p. This curve has a negative slope. (Since it is unlikely that a rational person would by more of a product the higher its price.) For the purposes of this assignment you can assume a linear demand schedule.

$q = m * p + b$, where m is the (negative) slope and b is the intercept on the q axis. The slope $m = -q_{max} / p_{max}$ where p_{max} is the maximum price the consumer is willing to pay, and q_{max} is the gluttony value, the amount the consumer can consume even if the product is mana from heaven (i.e., free).

The Simulation

The simulation is to run on the JADE platform. You only need one JADE container.

The Producer Agent

You need two Java classes (at least). One is the producer agent proper, the other is a GUI for the producer. (Call these Producer.java and ProducerGUI.java).

The GUI allows the user to set the name (just for decoration), unit price, and unit cost. The GUI should also allow the user to input the number of consumer agents that will be created. It also allows the user to see the total quantity sold and the amount of profit.

There are also two buttons. The first, when clicked, tells the Producer Agent to create the number of Consumer Agents chose by the user. This button should be disabled after one click. The second button should tell the Producer Agent to advertise the unit price and name of the product to the Consumer Agents. It should become enabled only after the first button is clicked.

The Producer Agent itself is responsible for creating the Consumer Agents, sending them INFORM messages containing price and name of product, and receiving a REQUEST message from each consumer requesting to purchase so much of the product.

The Producer Agent also sums these quantities purchased and uses the result to calculate the profit.

The Consumer Agents

The Consumer Agents are created by the Producer Agent. These agents do not have a GUI. The respond to an INFORM message containing price information form the producer. In response, the calculate a purchase quantity using their demand schedule functions, and send this information to the Producer Agent as a REQUEST message.

Ontology

The Producer and Consumer agents communicate using a special ontology as defined in EconOntology.java and associated files. These are packaged in c720a3Ontology.jar.

What to submit

Your files should be packaged in a jar file with the package structure cps720.assignment3 and cps720.assignment3.ontology, the latter containing the supplied ontology. Also include a readme.txt file describing your Java files. Make sure all group names are on all source files. Call the jar file cps720a3.jar.

Email the jar file as an attachment to david.grimshaw@sympatico.ca. Make sure all the group names are listed in the body of the email.

Exam Readme Fall 2000

Last Year's Exam

Fall 1999 Exam (Word 97 Format)

Fall 1999 Exam (HTML format)

Fall 2000 Exam

The exam is divided into three parts.

Part 1 consistes of 12 multiple choice questions worth 2 marks each.

Part 2 contains 5 "short answer" questions of which you are to answer two, worth 6 marks each. Some of these questions involve short essays, some calculations and some programming.

Part 3 contains one question, an Aglet program. This program is not unlike those in assignments 1 and 2.

What to study

The exam is based on the notes and the assignments. Links to other sites are intended as supplementary background material.

If you look at the index page for cps720 you see there are several subsections. Here are some highlights.

1. Ferber, Lange and Oshima on Agents. Characteristics of "agenthood".
2. Situated agents (really robotics) This is AI. The 4 levels of agent
3. The Aglet API. Aglets are communicative agents. Know the basics, Aglet,. AgleProxy, AgletContext, Message.
4. Agent communication. Basic facts about speech acts. Multi-agent architectures (e.g. InfoSleuth). You will not be asked about KQML.
5. XML. Relation with DTD. Read simple DTD, read/write XML. The two kinds of parser, DOM, SAX. (You will not be asked to use theSAX or DOM API's nor about XSL.)
6. Agent negotiation. Zero Sum games, prisoner's dilemma, problem of dishonesty and defection.

Remember, highlights are just that. You are responsible for the whole course!

Test Response Sheets

You will be given your test response sheet in class, or you can pick them up from Camille in V331. Take good care of it and bring it to the exam together with an HB pencil.

[top] [previous] [next] Questions?

Ryerson Polytechnic University

School of Computer Science

Final Examinations, Fall 1999

CPS 720 Artificial Intelligence Topics

Examiner: D. Grimshaw Time: 2 hours Closed Book

Part 1*. Answer any* ***four*** *(4) questions* (9 marks each).

1. What is an agent? Discuss from different perspectives.
2. What is an ontology? Define and discuss in relation to inter-agent communication.
3. Classify agents into 4 categories and discuss the Wall Following Agent in terms of one of these categories.
4. Describe the Aglet Mobile Agent Model. Discuss the principal components and mechanisms of Aglets.
5. Describe the InfoSleuth agent architecture.
6. Briefly describe the basic components of XML. Include a discussion of the DTD and DOM in your answer. To illustrate your answer, create a short XML file including its DTD.
7. Autonomous Internet agents will no doubt have to negotiate deals among themselves. In this case, the problem of deception arises. Discuss this problem in the context of Game Theory.

Part 2. Aglet Program (14 marks)

1. Write an Aglet program which has two Aglets, a master and a slave. The master creates the slave and dispatches it to a remote server. Upon arrival, the slave writes a message to the console of the remote machine saying "Greetings from <your name>". The slave then sends a message back to its master, saying "Your wish has been performed, O Glorious Master!", and then disposes of itself.

CPS 720 Resources and References

Agents in general

The UMBC AgentWeb

A central reference point for everything about agents.

Internet Resources

CERN School of Computing Aglets Course 1999

This is a webcast of parts of the 2 week summer course sponsored by CERN in held in Poland, starting Sept. 13 1999. The webcasts include video, sound and slides. A number of the lectures involve Java Mobile Agents using IBM's Aglets API. Given the time zones, try afternoon or evening. You will probably need rogers@home or Sympatico High Speed Edition to view them properly.

CERN Agent Course

Aglet Resources

The Aglets Home Page (IBM Japan) The original Aglets page

The Aglets Portal (UK)

Open Source Aglets

(Source Forge) - beware, you could get hooked on this :)

The Aglet API Documentation (local)

JADE

JADE Home Page

JADE Documentation (on SCS server)

The Foundation for Intelligent Physical Agents (FIPA) Home Page

Agent Cities

Ascape

Ascape Home Page

Some other agent systems

FIPA-OS

Zeurs

D'Agents

Agents in Distributed Systems

On the Structure of Distributed Systems, The Argument for Mobility

Todd Papaioannou's PHD Thesis. (PDF format - use Adobe Acrobat)

XML

Apache XML Project

- Xerces API Docs

The XML Portal

IBM's XML

Sun's XML

Robert Cover's XML and SGML Page - with introductions and FAQs

The Java /XML Tutorial from Sun

IBM XML Parser for Java

Microsoft XML

Books

AI Books

Nils Nilsson, Artificial Intelligence, A New synthesis, Morgan Kaufmann, 1998

Stuart Russell & Peter Norvig, Artificial Intelligence, A Modern Approach, Prentice-Hall, 1995

Jacques Ferber, *Multi-Agent Systems*, Addison-Wesley, 2000

The Aglet Book

Danny Lange & Mitsuro Oshima, Programming and Deploying Java Mobile Agents with Aglets (course text).

The code from Lange and Oshima's book.

XML Books

Brett McLaughlin, *Java and XML*, O'Reilly, 2000

Rich Eckstein, *XML Pocket Reference*, O'Relly, 1999

Hiroshi Maruhama, Kent Tamura, Naohiko Uramoto, *XML and Java*, Addison-Wesley, 1999

Games Theory Books

J. D. Williams, *The Complete Strategyst*, Dover, 1986

Steven Brams, Negotiation Games, Routledge, 1990

Robert Axelrod, The Complexity of Cooperation, Princeton University Press, 1997

William Poundstone, Prisoner's Dilemma, Doubleday, 1992

Jeffrey Rosenschein & Gilad Zlotkin, *Rules of Encounter*, MIT Press, 1994

Other Background Readings

On the Structuring of Distributed Systems:

The Argument for Mobility

By

Todd Papaioannou

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy

of Loughborough University
February 2000

For Jo

Acknowledgements

Undertaking a course of study that leads to the award of PhD is much like a journey of exploration and discovery. Although you may have some idea of where it is you wish to end up, the many rich experiences and pitfalls along the way are largely unforeseen. It is certainly an experience that I would recommend to anyone who believes they are capable. That is not to say, however, that it is a course suitable for everyone. The road to travel is long and tough, and many fall by the wayside.

My own journey has been one of academic learning and self-discovery. During my course of study, I have enjoyed incredibly the process of scaling new heights of knowledge, of cutting a trail where others may have never been, of using and pushing my mind to attack and answer the big questions. During this journey my mind has been refined to a sharpness and focus hitherto unforeseen to me, and I feel I am now able to wield my mind as a tool, in all situations. This has allowed me to look within, and understand exactly whom I am. In addition, my character has grown and expanded with a wealth of new experiences that have served to polish it.

I feel lucky to have undertaken my research in a relatively new field, where the boundaries and rules have not been defined yet. This has afforded me an academic freedom that many students do not enjoy, and allowed me to follow an academic path out of the reach of many. This type of work cannot be done alone in isolation though, and I would like to take this opportunity to thank those who have made it possible for me to get this far.

Firstly I would like to thank all my family, especially Jill, Les and Yannis, for their continued support throughout my many years of study. Without their help, I would have been unable to complete my work. I hope my completion goes some way to repaying their trust and support.

To study for a PhD requires a suitable environment and support in which to do so. Most important in providing this has been my supervisor Dr. John Edwards. John deserves special credit for having the patience to guide a determined and unconventional student, even when many of the proposed ideas were contrary to his own philosophies. I am sure the experience must have been challenging, but I believe

that we have both learnt greatly from it. I would also like to thank the other members of the MSI Research Institute for providing a stimulating social environment, and in particular Ian Coutts and Paul Clements for their support in hearing and critiquing my research philosophies as they developed.

In addition, I would like to offer my thanks to many people around the world who have had some input or influence over the course of my study. In particular, my friend and colleague Nelson Minar, who has been a trusted source of advice throughout the journey, and Dr. Danny Lange who has been an excellent mentor and font of wisdom. Also, the members of the agents, mobility and dist-obj mailing lists have provided an invaluable service as a community of peers amongst which to discuss my research. Many of the ideas expressed in this thesis have been shaped and refined in those forums.

One cannot work on anything exclusively for so long and so hard, without the need for respite. I have many friends to who I owe thanks, who have allowed me to relax, rage, or lose myself, away from grindstone. Some deserve special mention. Firstly, my best friend Darren May, who has been there from the early years and will be there at the end. Also, my friends Derek Woods and Andy Grant who have been my partners in many misdemeanours at Loughborough through the years.

Lastly, but most importantly I would like to thank my partner, Joanna Henderson, whose unswerving love, support and companionship have allowed me to concentrate my efforts on achieving my goals. She truly is a wonderful person and I count myself extremely lucky to be with her.

Abstract

The last decade has seen an explosion in the growth and use of the Internet. Rapidly evolving network and computer technology, coupled with the exponential growth of services and information available on the Internet, is heralding in a new era of *ubiquitous computing*. Hundreds of millions of people will soon have pervasive access to a huge amount of information, which they will be able to access through a plethora of diverse computational devices. These devices are no longer isolated number crunching machines; rather they are on our desks, on our wrists, in our clothes, embedded in our cars, phones and even washing machines. These computers are constantly communicating with each other via LANs, Intranets, the Internet, and through wireless networks, in which the size and topology of the network is constantly changing. Over this hardware substrate we are attempting to architect new types of distributed system, ones that are able to adapt to changing qualities and location of service. Traditional theories and techniques for building distributed systems are being challenged. In this new era of massively distributed computing we require new paradigms for building distributed systems.

This thesis is concerned with how we structure distributed systems. In Part I, we trace the emergence and evolution of computing abstractions and build a philosophical argument supporting mobile code, contrasting it with traditional distribution abstractions. Further, we assert the belief that the abstractions used in traditional distributed systems are flawed, and are not suited to the underlying hardware substrate on which contemporary global networks are built. In Part II, we describe the experimental work and subsequent evaluation that constitutes the first steps taken to validate the arguments of Part I.

The experimental work described in this thesis has been published in [Clements97] [Papaioannou98] [Papaioannou99] [Papaioannou99b] [Papaioannou2000] [Papaioannou2000b]. In addition, the research undertaken in the course of this PhD has resulted in the publication of [Papaioannou99c] and [Papaioannou/Minar99].

Contents

List Of Tables

List of Figures

Preface

Mobile Code is a new and generally untested paradigm for building distributed systems. Although garnering many plaudits and continually increasing in popularity, the technology and research field remain relatively immature. So far, most research has focused on the creation of mobile code frameworks, and as yet, there is no conceptual framework with which to contrast results. Equally, there is no clear understanding of the new abstractions offered by this paradigm. Further, many conclusions drawn about the technology remain qualitative and subjective. This dearth of quantitative results means as yet it has not been possible to evaluate the potential of both the technology and the paradigm.

It is against this backdrop that the work described in this thesis has been conducted. Before an accurate and informed decision about the suitability of mobile code technology can be made, a fuller appreciation of the paradigm is required. It is the author's opinion that the central essence of a new paradigm is the abstraction it offers to the designer. Therefore, the contribution of this thesis addresses the issues of *understanding* and *evaluating* the design abstractions offered by mobile code.

The first part of this thesis is concerned with building an *understanding* of the abstractions offered by mobile code, and the implications of using them. Certainly, it would be impossible to undertake this research without a context within which to analyse the new paradigm. To this end, we trace the emergence and evolution of abstractions employed throughout the history of computing, in an attempt to understand the reasons behind the existence of contemporary traditional distribution abstractions. We also build a philosophical argument supporting mobile code, contrasting it with traditional distribution abstractions. Further, we assert the belief that the abstractions used in traditional distributed systems are flawed, and are not suited to the underlying hardware substrate on which contemporary global networks are built.

In chapter one, we review the history of computing, and the abstractions that have been employed within this field. We begin our journey by examining the early years of computing, and trace the consecutive developments that have shaped the evolution of our present day computing landscape. We build a picture of the key phases in this

evolution, and the gradual layering of abstractions, one atop another, that characterises evolution in this area.

In chapter two, we return to focus more directly on the emergence of distribution. In examining today's distribution mechanisms we show that the fundamental abstraction in these systems is one of *location transparency*. The chapter demonstrates that the emergence of location transparency is a result of the layers of abstraction found beneath it. We argue that by using the location transparency abstraction we are attempting to impose an unsuitable abstraction onto the underlying computational substrate.

In chapter three, we begin our examination of the new design abstractions offered by Mobile Code. We discuss what makes mobile code systems different from contemporary ones and characterise these new abstractions as embodying *local interaction*. Finally, we argue that by employing this new paradigm we are using an abstraction more wholly suited to the underlying computational substrate, and thus to building distributed systems. This chapter concludes our philosophical argument concerning the structuring of distributed systems.

The philosophical argument built in Part I is extensive, and a full experimental investigation is beyond the scope and timescale of a PhD. Therefore, in Part II we take the initial steps required to validate the arguments expressed in Part I. If Part I was concerned with understanding the mobile code abstraction, then Part II is concerned with *using and evaluating* it. The experimental work is conducted by applying the new paradigm to a real world manufacturing system application, based on data derived from an industrial case study.

In chapter four, we present the rational for the experimental research undertaken in this thesis, and describe how it will support the arguments made in Part I. Further, we describe the technical issues involved with implementing mobile code abstractions, and discuss some of the advantages claimed for this new technology. Lastly, we review several of the better-known mobile code frameworks available to researchers, before presenting IBM's Aglet Software Development Kit, the framework used in our experimental work.

In chapter five, we describe a case study undertaken in the UK. The case study has been used to generate a real-world model of the Sales Order Process (SOP) of a manufacturing enterprise that is used in the subsequent implementation work. In addition, several requirements of the company were identified which will be used in later chapters as “scenarios for change” with which to test and measure our experimental implementations.

In chapter six, we describe the creation of two prototype mobile code systems. Their common parts and differences are discussed, along with the supporting tools that have been created.

In chapter seven, we begin our evaluation of the two prototype systems. Firstly, we describe the process through which we have generated several tangible software metrics. We then evaluate the prototypes through the “scenarios for change”, and reflect on what has been learnt.

In chapter eight we conclude the research undertaken in this thesis, and discuss the implications of the work, and avenues for further investigation.

Part I

Understanding

1 Abstraction

1.1 Introduction

Computers are fulfilling an increasingly diverse set of tasks in our society. They are silently assuming many mundane but key tasks, providing seamless assistance to support our lifestyles. They control our car engines, our environmental climate and even our toasters. Increasingly, sophisticated hardware is the supporting substrate for increasingly complex software. Yet despite major advances in our understanding of the construction of software, building flexible and reliable systems remains a considerable task. Increasingly powerful abstractions are employed by software engineers in an attempt to reduce the cognitive complexity of such tasks.

The emergence of computing abstractions has been instrumental in defining today's computing landscape. To fully understand its present day shape, we must first understand the forces and issues that influenced its evolution. This chapter presents a brief history of computing and the levels of abstraction developed and employed within this field, and discusses the emergence of each abstraction.

1.2 A Brief History of Computing Time

> "In the beginning there was binary. And 'lo, von Neumann did say 'that's too damn tough to understand! Can't we make it any simpler?'"

In the 1940's, the mathematician John von Neumann pioneered research into formalising the basic architecture for a computing machine. The Von Neumann architecture specified a computer in terms of three main components:

- **A Memory**: a large store of memory cells that contain data and instructions
- **An Input/Output unit**: to enable interaction and feedback with the user
- **A Central Processing Unit (CPU)**: responsible for reading and writing instructions or data from the memory cells or from the I/O unit

During execution, the CPU takes instructions and data from the memory cells one at a time, storing them in local cells known as registers. The instructions cause the CPU to manipulate the data via arithmetic or logic operations, before assigning any results back to memory. Thus, the execution of instructions results in a change in the *state* of

the machine [Burks46]. The three components of a computer are able to interact via a communications bus (see Figure 1).

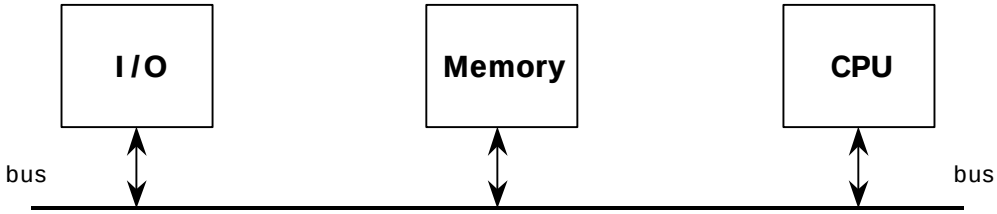


Figure 1. The von Neumann Computer Architecture

Von Neumann's research was based on the earlier theoretical work of Church and Turing on state machines [Church41] [Turing36]. Importantly though, it established a hardware architecture for a computing machine that would serve as a reference platform for decades to follow. Although we are generally accustomed to thinking of computers as extremely complex machines, the central architecture itself is quite simple. At the most basic level Harel states:

> "A computer can directly execute only a small number of extremely trivial operations, like flipping, zeroing, or testing a bit" [Harel87]

Nonetheless, von Neumann had taken the first step along a long path of evolution that would culminate in the computer systems we take for granted today. This evolution could not have taken place without advances in hardware design and manufacture, however, for the scope of this thesis we are interested only in the abstractions and technologies that have evolved to support the construction of software.

Since its creation, the von Neumann architecture has fundamentally influenced the way we think about and build our computing systems. Most contemporary programming languages can be viewed as abstractions of the underlying von Neumann architecture. These languages retain as their computational model that of the von Neumann architecture, but abstract away the details of execution. The sequential execution of language statements (instructions) changes the state of a program (computational machine) through assignment and manipulation of variables (memory cells). These languages, known as *imperative languages*, have developed through the addition of layers of increasingly high levels of abstraction [Ghezzi98]. In the next section we examine the emergence and evolution of imperative languages,

and discuss the ascending tower of abstractions that we use to construct software systems.

1.3 Procedural Abstractions

Programming a computer to perform a particular task in the early years of computing was extremely difficult and time consuming [MacLennan87]. The von Neumann architecture provided a computational model that programmers could use to manipulate physical memory locations. Nevertheless, this was still an arduous task, as each memory location was identified by a long binary string. Humans do not naturally think in binary, and programming in this manner was not only complex but also prone to error [Hopper68].

To alleviate the inherent difficulties with working in binary a new family of languages, known as assembly languages [Harel96], were developed. Assembly languages served as a primitive form of abstraction, which masked the architecture of the underlying hardware. With this new abstraction, programmers were able to specify memory locations symbolically, rather than with an unwieldy binary string.

The creation of assembly languages was the next step towards unlocking the full potential of the computer. Using them, programmers were no longer concerned with the location of individual registers and memory cells. They were able instead to program with symbolic representations of their computing machines. From here, it was a relatively simple matter to begin constructing repeatable computing algorithms from assembler symbols [Wexelblat81]. These algorithms became a layer of abstraction above the assembly symbols, which themselves were a layer of abstraction above the hardware. Quickly, the pattern for computing evolution had been defined: it would evolve through the gradual layering of ever subtler and complex levels of abstraction. Each layer abstracting away the minutiae whilst retaining as their underlying computational model the von Neumann architecture. Figure 2 shows the abstractions of assembly languages, and then computing algorithms layered over the underlying von Neumann computational model.

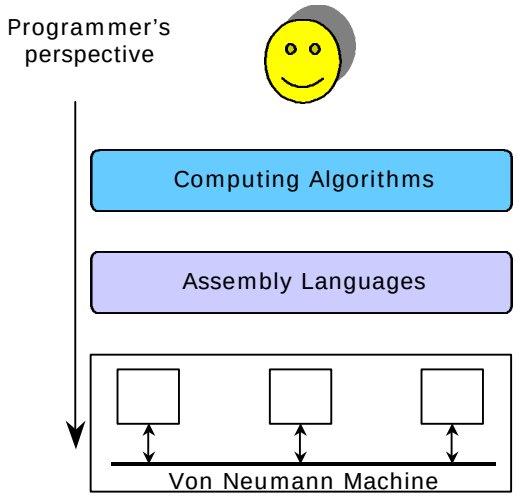


Figure 2. Early Layers of Abstraction

These early layers of abstraction were a considerable improvement in the way computer programs were constructed. However, even more significant improvements in the usability of computers would occur with the arrival of programming languages.

A programming language is a formal notation for describing algorithms for execution by a computer [Ghezzi98]. They provide abstractions to overcome the complexities involved in constructing a software program, so that a programmer does not need to be capable of manually producing the many machine level instructions that are required to get a computer to perform a particular task. The first types of programming languages developed were known as pseudo code languages.

Pseudo codes arose because in some instances programmers found that the hardware specific instructions available on their particular computing architecture were not sufficient to support the range of operations they required. Pseudo codes are machine instructions that differ to those provided by the native hardware on which they are being executed. They are invariably executed within an interpreter [MacLennan87], a software simulation of a computational machine, a virtual machine, whose machine language is the pseudo codes. The virtual machine would normally offer facilities that were not available in the real computer, for example, new data types (e.g. floating point) or operations (e.g. indexing). Ergo, pseudo codes added yet another, higher layer of abstraction, and were the initial steps taken in moving towards a tool that allowed a programmer to construct software in a language that bore no resemblance to its machine code representation [Hopper68]. Unfortunately, pseudo code languages

were hampered by slow execution speeds, since the interpreter had to first convert the codes to native instructions prior to execution. To overcome this inefficiency a new tool known as a compiler was produced. A compiler is a computer program that translates programs specified in high-level languages, for example pseudo codes, into the native hardware's assembly language [Harel93]. The program need only be translated once, but could be executed at native speeds many times, which was a distinct advantage over programs that had to be interpreted every time.

The advent of compilers led to the creation of new programming languages, known as 1st generation languages. The best known of these are IBM's Mathematical FORmula TRANslating system (FORTRAN) [IBM56], COmmon Business Oriented Language (COBOL) [DoD61], and ALGOrithmic Language (ALGOL) [Perlis58] which appeared in the mid to late 1950's respectively. These languages allowed a programmer to use a mathematical notation in order to solve a problem. FORTRAN and ALGOL were defined as tools for solving numerical scientific problems, those that required complex computations on relatively simple data, for example simulating numerically the effects of a nuclear reaction. COBOL was developed as a tool for solving business data-processing problems, those that required computations on large amounts of structured data, for example a payroll application. It was able to satisfy the needs of the bulk of the applications of the day, and its success has meant it remains in use over thirty years after its introduction [Wilson93].

The advent of compilers and 1st generation languages meant it was possible to develop computer programs without any knowledge of how your program was actually transformed into the native instruction set required by the machine upon which it was intended to execute; the translation was automatically performed by the compiler. One of the most important concepts embodied in the abstractions offered by 1st generation languages was the separation of a program into two distinct parts. The description of the data contained within the program was known as the *declarative* part, and the program logic that controlled the execution of the program and manipulation of the data was known as the *imperative* part.

Once begun, the development of programming languages progressed rapidly, and soon 2nd generation languages would emerge. These new languages were generally descendants of 1st generation languages, influenced by the lessons learnt in the early

years. They are characterised by offering a much higher level of structured flow control to the programmer whilst simultaneously introducing new techniques to aid the composition of computer programs. Typical of this set of languages is ALGOL 60 [Naur63]. The product of a committee, ALGOL 60 introduced major new concepts such as syntactic language definition [Backus78], the notion of block structure [Wilson93] and recursive programming [Ghezzi98]. Further improvements to structured flow in languages such as loops, conditional statements, sequential constructs and subroutines [Harel93] meant that some of the hardware-influenced instructions prevalent in 1st generation languages, such as the infamous GOTO[1] statement [Dijkstra68], could be removed.

By the 1970's it was becoming clear that the need to support reliable and maintainable software had begun to impose more stringent requirements on new programming languages [Ghezzi98]. Programming language research in this period emphasised the need for eliminating insecure programming constructs. Among the most important language concepts investigated in this period include: strong typing [Cardelli85], static program checking [Abadi96], module visibility [Parnas72a], concurrency [Ben-Ari90] and inter-process communication [Simon96]. Greater significance was now placed on building reliable software, and the term *software engineering* [Naur68] was used to describe an emerging methodology for dealing with the full lifecycle of software development, from specification to production. In general, it is fair to say that 3rd generation languages built on the previous generation by working at improving the software engineering principles inherent, and enforced by the languages. Some important examples of 3rd generation languages are Euclid [Lampson77], Mesa [Geschke77] and CLU [Liskov81]. The development of these languages was directly influenced by the need to improve systems programming [Wilson93], the creation of operating systems and tools such as compilers, and to produce verifiable programs.

In the last half of the 1970's new languages such as Pascal [Jensen85] [ISO90b] and C [Kernighan78] were developed. Both offered the programmer power, efficiency, modularisation and availability on a wide array of platforms. With Pascal though, Wirth aimed to create a language that would also be suitable for teaching

1 Strangely, the much maligned GOTO statement continues to exist in many languages

programming as a logical and systematic discipline, thus encouraging well-structured and well-organised programs. C on the other hand combines the advantages of a high level language with the facilities, flexibility and efficiency of an assembly language. However, to ensure the degree of flexibility required by systems programmers C does not include type checking, meaning that it is much easier to write erroneous programs in C than in Pascal [Wilson93]. Both languages continue to be widely and successfully employed today.

1.3.1 Commentary

When von Neumann first specified his computing architecture, he set the direction in which our computing landscape would evolve. Since then, we have evolved through the gradual layering of increasingly powerful abstractions upon each other. The progressive development of programming techniques that ascended via early unwieldy bit strings, through assembly mnemonics, pseudo codes, compilers and three generations of programming languages signified the first phase of our evolution. In this phase programmers were gradually lifted out of the mire, and spared the task of remembering the location of each cell or register they wish to use. They were now able to specify programs in powerful and efficient languages, without requiring any hardware specific knowledge of the computer they were using. By progressively exploring and building up the layers of abstraction, the computer had been transformed from a slow and cumbersome behemoth to a powerful, flexible tool.

In this thesis we term this period of computing the *procedural abstraction* phase. It is characterised by the development of new computing abstractions and new techniques for controlling program structure and flow. Figure 3 illustrates the individual layers of abstraction discussed in the previous section. Each box roughly represents the beginning of each abstraction, and is intended to depict the progressive layering of abstractions as programming languages were developed. Certainly each box should not be interpreted as a finite lifetime for each abstraction. For example, assembler continues to be heavily used in modern military aircraft systems [Bennet94].

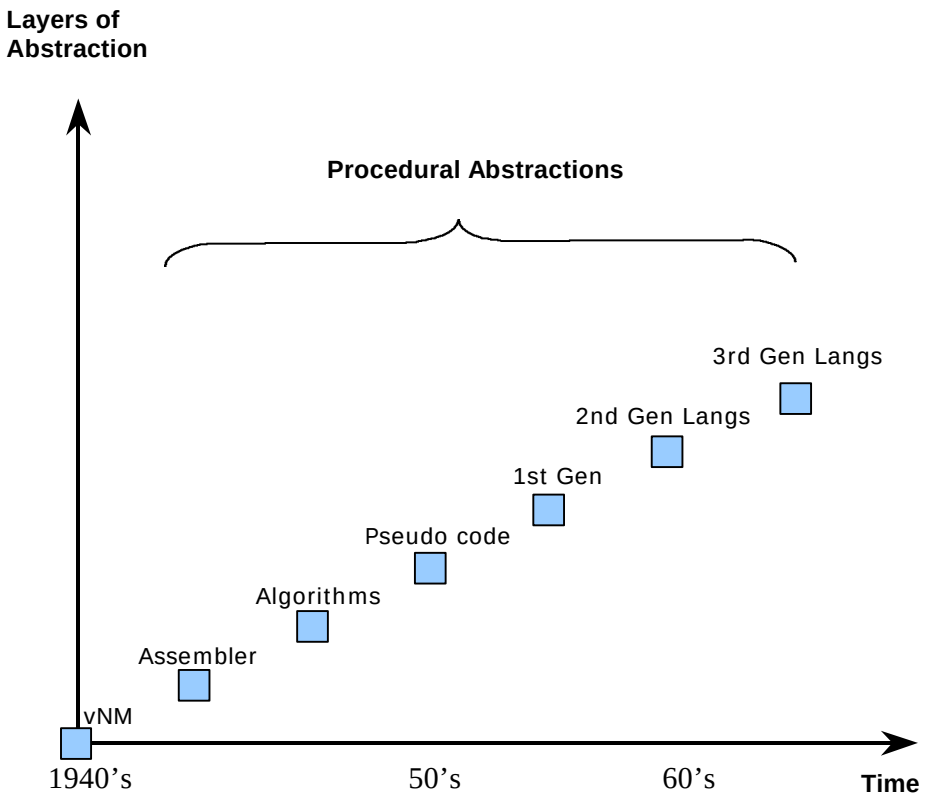


Figure 3. The layers of abstraction in the Procedural Abstraction Phase

1.4 Programming Abstractions

> "Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious." [Raymond98] citing and re-interpreting [Brooks95]

The mid to late 1970's saw a new trend develop within the world of computing. Supported by more powerful tools and languages programmers began to build increasingly large and complex programs [DeRemer76]. These programs were no longer standalone edifices, capable of performing a single task. Rather, they were *systems*, capable of a multitude of tasks.

The sheer size of these systems meant that for reasons of clarity and maintenance it was becoming increasingly important to organise programs into discrete modules [Knuth74]. With the Modula-2 language [Wirth77], Wirth attempted to extend Pascal

with modules and while not wholly successful the experiment was an indication of the possible advantages [Wilson93]. Language researchers soon realised that it was not only advantageous to separate programs into discrete modules, but also to conceptually encapsulate data and logic within larger entities. Such encapsulations were known as abstract data types [Hoare72] and enabled the programmer to specify new data types in addition to those primitives already supported by the language. For these new abstractions, programmers could define operations through which they could be manipulated, while the data structure that implements the abstraction remained hidden. Information or data hiding [Parnas72a] ensures that the internal data of a new type will only be manipulated in ways that are expected. The late 1970's and early 80's saw an explosion of new programming abstractions, such as type extensions [Wirth82], concurrent programming [Andrews83] and exception handling [Goodenough75]. Again, the motivation was to make software more maintainable in the long term. A resulting synthesis of many of these new techniques is the language Ada [DoD80], which can be viewed as the state-of-the-art for that time.

The 1980's saw the arrival of Object-oriented Programming (OOP), the origins of which can be traced back to Simula 67 [Birtwistle73]. An object is an encapsulation of some data, along with a set of operations that operate on that data. Operations are invoked externally by sending messages to the object [Blair91]. Thus, each object is an abstraction that both encapsulates and acts upon its logic and data respectively. This allows a programmer to view their system as being composed of conceptually separate entities, or objects. The OOP abstraction also builds on the previously discussed advances in modularity, data abstraction and information hiding, by including facilities for software reuse [Ghezzi98]. Newly created objects in the system are not implemented from scratch, rather they may inherit pre-existing behaviour from a parent object, and implement only the required new behaviour. OOP initially became popular through the success of Smalltalk [Goldberg83], but was more widely accepted with the advent of C++ [Stroustrup92], an extension of C. Other popular OO languages include Dylan [Apple92], Emerald [Raj91], Modula-3 [Nelson91] and more recently Java [Gosling96].

1.4.1 Commentary

In this thesis we term this ascendance from building programs, to architecting systems as the *programming abstraction* phase. It is characterised by the development of new techniques for modularity, data abstraction and software reuse, and would result in systems that were easier to change and maintain [DeRemer76] and were more reliable [Horowitz83]. In Figure 4 below we see the programming abstraction phase continue the gradual layering of abstractions.

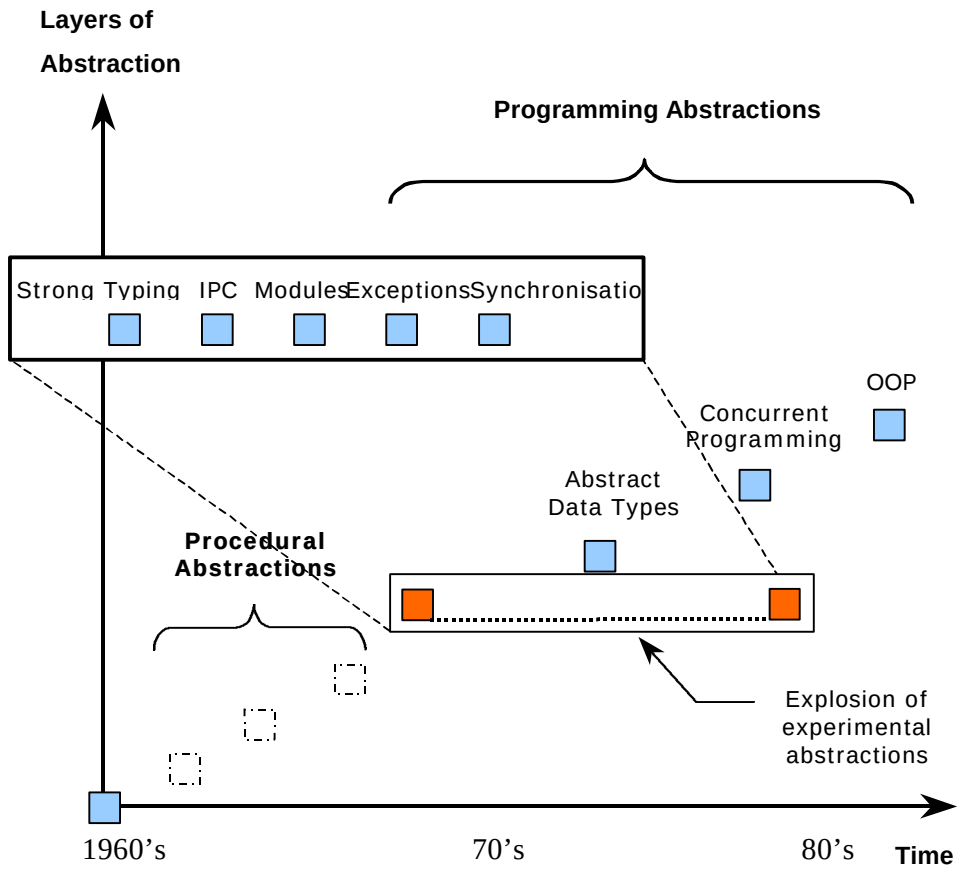


Figure 4. Layers of abstraction in the Programming Abstraction Phase

1.5 The Far Side

So far, we have concentrated solely upon the ascending layers of abstractions that are present and supported by imperative or procedural languages. These languages employ the von Neumann architecture as their underlying computing model, and are greatly influenced by the necessity for efficient execution.

With the decreasing costs of computer hardware, however, radically different designs of computing machine have become possible. This has opened up the possibility that other computational models could be found, and that it may be possible to design the computer hardware to fit the model, rather than the other way round [Wilson93]. As early as the 1960's there have been attempts to define programming languages whose computational models were based upon well-characterised mathematical principles, rather than on efficiency of implementation [Ghezzi98]. These alternative camps can be split into functional and logic programming languages.

Functional languages use as their basis the theory of mathematical functions, and they differ greatly from imperative languages as they do not support the concept of variable assignment. Assignment causes a change in value to an existing variable, whereas the application of a function causes a new value to be returned. This has important implications for the problem of concurrency, since in an imperative language it is possible to refer to a variable or object that has been reassigned without your knowledge. In a functional language, a function may be called at any time, and will always return the same value for a given parameter [Hudak89]. Further, since variables cannot be altered by assignment, the order in which a program's statements are written and evaluated does not matter; they can be evaluated in many different orders. Thus, programs can be modified as data and data structures can be executed as programs. The key concept in functional programming is to treat functions as value, and vice versa [Watt96].

The archetypal functional programming language is generally considered to be LISP [McCarthy60], which was developed in the late 1950's. It is based upon the theory of recursive functions and lambda calculus, work that was developed in the early 1940's by Church [Church41]. Since its creation LISP has become one of the most widely used programming languages for artificial intelligence and other applications requiring symbolic manipulation [Pratt84], for example symbolic differentiation, and has spawned a plethora of individual dialects. As with the imperative camp, there have been several other implementations of functional languages during the following years, for example APL [Iverson62], ML [Milner90], Miranda [Turner85] and Haskell [Thompson96]. Latterly, the competing dialects of LISP were unified in Common LISP [Bobrow88].

Another variant in the field of programming languages are those defined as logic programming languages. The main difference between functional and logic programming languages is that programs in a pure functional programming language define functions, whereas pure logic programming defines relations [Ghezzi98]. Logic programming languages first appeared in the late 1970's and are based on the principles of first order predicate calculus [Mendelson64] and eschew all relation to the underlying machine hardware. In contrast to other styles of programming, a programmer using a logic language is more involved in describing a problem in a declarative fashion than in defining details of algorithms to provide a solution [Callear94]. The knowledge about a problem and the assumptions about it are stated explicitly as logical axioms [Kowalski79]. This problem description is then used by the language's computational machine to find a solution. To denote its distinctive capabilities, in this case a computational machine that can execute a logical language is often referred to as an inference engine. Synonymous with logic programming, and the ancestor of all logic languages is PROLOG [Clocksin87].

1.5.1 Commentary

The genres of functional and logic programming languages are an important contribution to our computing landscape. Both are declarative languages and are characterised as being independent of the underlying hardware upon which they are executed; they are abstractions that are not influenced by the von Neumann architecture. However, to achieve this independence efficiency has been sacrificed [Wilson93]. This, and the fundamental change of programming mindset required for those accustomed to the imperative style has been detrimental to their widespread acceptance and deployment outside of the artificial intelligence and expert systems communities.

Perhaps most revealing in the functional vs imperative language debate is the 1978 Turing Award lecture given by John Backus [Backus78]. In this, and his paper, Backus argues that conventional programming languages are fundamentally flawed in their design since as they are inherently linked to the underlying von Neumann architecture. Backus goes on to demonstrate the advantages of functional languages over imperative ones, and further introduces a new functional language, FP. His

assertion is that the underlying abstractions we use are important, and can affect the way we think, use and build computer systems and software.

1.6 Conceptual Abstractions

In the last decade, software engineering has been scaling new heights of abstraction. Program development has undergone a tremendous revolution; in the way that programs are entered into the computer, and the way programs are assembled from existing parts [Ghezzi98]. Programmers are now able to use integrated development environments and libraries of predefined modules to rapidly compose software systems visually [Zak98].

Recent developments such as Components [Sun97] allow developers to view their systems with a larger granularity than objects. Components may be large, for example a Request Broker consisting of hundreds of objects, or as a small as a GUI widget consisting of only a few objects. In addition, techniques such as Software Patterns [Gof93] enforce a rigid literary methodology for expressing the essence of a recurring software abstraction. A pattern may be viewed as a monograph on the particular abstraction, and describes the many facets required to consistently select and use an appropriate abstraction, what issues are involved and when not to use this pattern. It is a distillation of knowledge gained by many experts over the years. Aspect Oriented programming [Kiczales97], Actors [Agha97], and Agent Oriented Programming [Wooldridge99] are examples of techniques that attempt to remove any notion of hardware from the abstraction. In fact, one may view them as attempts to personify software. In particular, the autonomous agent community appears to be having much success with its approach, allowing designers to view and build systems in a new manner, with new perspectives [Jennings et al98].

These new abstractions are no longer merely based on technological developments in language or compiler design. They are conceptual abstractions, allowing the software designer to view their system at a level completely removed from any of the underlying hardware issues. Figure 5 is the culmination of this chapter’s examination of the gradual layering of abstractions. It illustrates chronologically all three phases of abstraction we have identified: procedural, programming and conceptual, and how each individual abstraction has been layered over those preceding it.

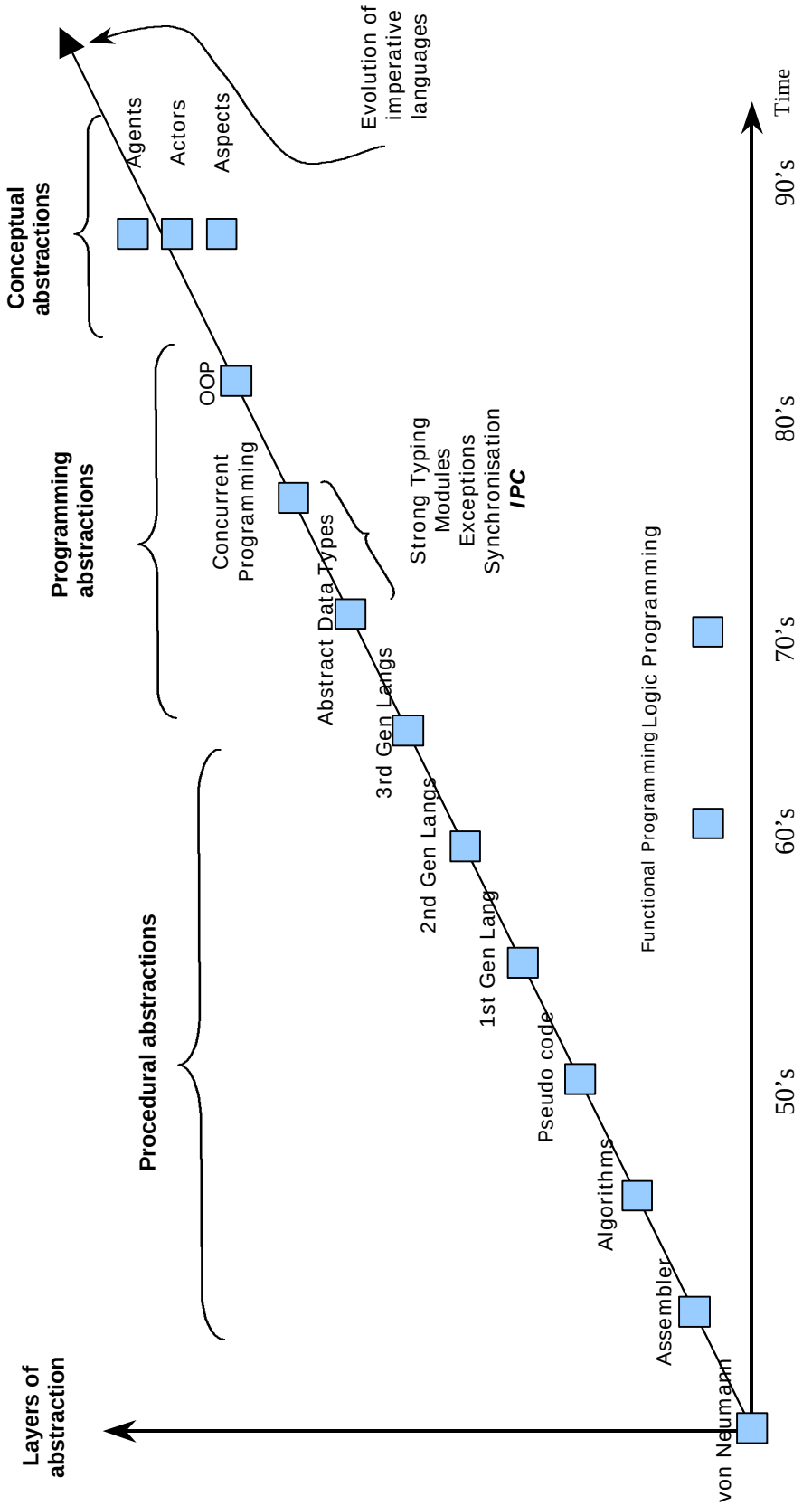


Figure 5. The full Layers of Abstraction diagram

1.6.1 Commentary

The computers we build are no longer merely high-powered calculating machines; they are useful tools that can be both incredibly flexible, and stubbornly inflexible at the same time. Our on-going affair with computers has been characterised by our attempts to harness their power, and apply them to ever more diverse situations. This affair has been tempered, however, by the complexity inherent in a computing system. The complexity involved has forced us to continually refine the languages and tools we use to build software systems. In our efforts to understand and use the technology we abstract away the details, pasting on ever more elaborate facades to hide us from the true complexities involved in creating software. Gradually we have layered increasingly complex abstractions over those lying beneath, until it is no longer even a requirement to be aware of those early abstractions. Modern day programmers have rapid development tools and libraries with which to build software. They employ conceptual abstractions that bear no resemblance to underlying hardware upon which their creations will be executed. These layers of abstraction mean that modern day programmers are not required to be aware of the abstractions that lie below, that they depend on to deliver their creation.

1.7 Concluding Remarks

> "Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more" [Backus78].

> "Appropriate abstractions and proper modularisation help us confront the inherent complexities of large programs" [Ghezzi98]

Abstractions are an immensely powerful tool. They allow us to manage the complexity of a situation, and to rationalise about it by removing those details we consider inessential. Further, as we attain understanding of complex issues, we construct additional layers of abstraction over those beneath, continually ascending. If we are to consider abstractions that exist within these layers we must understand the reasons for their existence, and the base abstractions that support the grand edifice.

This chapter has presented a brief history of our progress up the computing abstraction tower. It has examined the chronological development of computing

architectures and programming languages, and presented a brief explanation of their existence. Latterly, the discussion continued by examining more recent programming and conceptual abstractions and their position in the Tower of Abstractions. Although the functional and logic programming camps offer us a declarative alternative they are in the minority. The overwhelming majority of languages in use today are imperative. They are powerful abstractions whose roots are found in the pioneering work of John von Neumann in the first half of this century. Our computing evolution has been characterised and dictated by the von Neumann architecture. It has influenced the design of all imperative languages to follow, and therefore those abstractions subsequently attained by using the languages.

The aim of this thesis is to understand the mobile agent abstraction, a new technology and abstraction for building distributed systems. The review in this chapter has provided a context and history in which new and existing abstractions can now be reviewed. In the next chapter we examine the abstractions currently used in building contemporary distributed systems.

2 Towers of Babel

2.1 Introduction

In the 1970's, networking began to emerge as an important aspect of computer systems. Driven by applications in the military and airline industries, computer systems were connected and inter-operation became widespread [Cerutti83]. During the 1980's, distributed computing became a vital aspect of many computer systems. In the early 2000's, we are beginning to see the emergence of ubiquitous computing: characterised as a massive heterogeneous "sea" of disparate computational devices, with varying connection bandwidths and an ever-changing topology of connections [Weiser91].

This chapter examines the emergence of distribution and discusses the path of its evolution. In examining today's distribution mechanisms we show that the fundamental abstraction in these systems is one of *location transparency*. Further, we demonstrate that the emergence of location transparency is a result of the layers of abstraction found beneath it. We argue that by using this approach we are attempting to impose an unsuitable abstraction onto the underlying hardware substrate.

2.2 The Advent of Distribution

Before the invention of computers, processing information was both slow and tedious [Rose90]. The advent of computers has transformed the world, and the way in which we work with information [Simon96]. However, using and storing this information in isolation, like any expensive resource, is inefficient [Peters85]. Ergo, unless our computers are to exist in isolation, we require methods that allow computers to meaningfully interact [Cerutti83], and ways of transferring information between them. Communication networks, which interconnect computers and allow them to work in concert, are a common solution to this problem [Sloman87].

However, merely physically connecting computers is not enough to achieve logical interaction in its own right. Computers must adhere to a common set of rules or protocols for defining their interactions [Rose90]. By connecting separate computers, we make it possible for the programs executing on those computers to interact. When

processes on separate computers interact, we term the whole a distributed system. In the next section, we examine the software architectures used in building networks, which ultimately support any communication between networked computers.

2.3 Distributed Communication

> A network is an interconnected collection of two or more autonomous computers [Tanenbaum96].

Distributed computing as we understand it today is a far cry from the limited facilities of early distributed systems, such as remote job entry handlers [Boggs73]. Their role however was simple - to allow scarce and expensive information and resources to be shared by users. Ever since computer users began accessing central processor resources from remote terminals over 40 years ago, computer networks have become more versatile, more powerful and inevitably more complex [Green80].

At the heart of distributed computing are communication networks. They are the infrastructures that support information flow between computers. The initial development of such networks was fostered through experimental networks such as ARPANET [Roberts70] [Cerf74] and CYCLADES [Pouzin73]. ARPANET, which went live in December 1969, was initially motivated by the requirements of the US Military for a communications network that could survive a nuclear war [Tanenbaum96]. This early work established the procedures for connecting computers and facilitating their interaction. Just physically connecting computers was not sufficient to ensure successful interaction though. Two computers wishing to communicate must adhere to a common set of rules for defining their interactions. This rule set is termed a *protocol*, and is an agreement between the communicating parties on how communication is to proceed [Rose90].

To reduce their design complexity, network architectures are organised as a series of layers or levels of abstraction, each built upon the preceding one. Whilst the number and nature of these layers may differ between architectures, their purpose is similar: to offer services to the higher layers, shielding them from the details of how the offered services are actually implemented [Tanenbaum96]. Each layer has its own particular

communication protocol, and collections of protocols defined in terms of a common framework are known as a *protocol suite* or *stack* [Rose90].

In early computer systems, it was common for each application found on a computer to employ its own protocol stack. This communication support was usually built into the application, and was not available for use by any other applications. This approach therefore had the inherent disadvantages of duplicated functionality and inefficient resource usage. To alleviate this undesirable situation, research focused on providing communication mechanisms at the operating system level through the provision of shared communication suites [Sloman87].

Although a vast improvement, facilities provided by the operating system were invariably specific to the particular type of computer on which they were executing. In the mid 1970s, computer vendors began to develop their own network architectures, to enable communication between their own ranges of machines. Important examples of this period are the Internet model [Metcalfe76] [Comer91] that emerged from ARPANET [McQuillan77], IBM's Systems Network Architecture (SNA) [McFadyen76] [Cypser78] [Gray83] and Digital's DECnet [Wecker80] [Malamud91]. This meant however, that since each suite was developed for the vendors' own machines, they were usually composed of proprietary (closed) protocols. This situation posed two considerable problems:

- Systems from competing vendors were not able to interoperate
- The communication specification was controlled by a single organisation

Since the vendors controlled the protocol specification, they also had the power to change the specification at their discretion [Cerutti93]. Understandably, this made third party developers very nervous in adopting and working to a standard whose specification might be changed at any given moment. Although subsequent publishing of the protocol specifications aided their widespread adoption, the issue remained [Rose90]. Further, as each proprietary communication suite evolved, systems from competing manufacturers became even more incompatible.

The splintered evolution of incompatible communication suites forced the computing community to realise that standards were required to enable interaction between different types of computer [Mullender93]. In 1977, the International Standards

Organisation (ISO) began working towards defining a non-proprietary (open) suite of protocols. The resulting standard is known as the ISO Open Systems Interconnection (OSI) reference model [Zimmermann80] [ISO83] [OSI84] [STA87], and is jointly defined by ISO and the International Telecommunications Union (ITU-T)[2]. Most of the proprietary suites that preceded the OSI model have since undergone modification and are now considered as specialised incarnations of the OSI model.

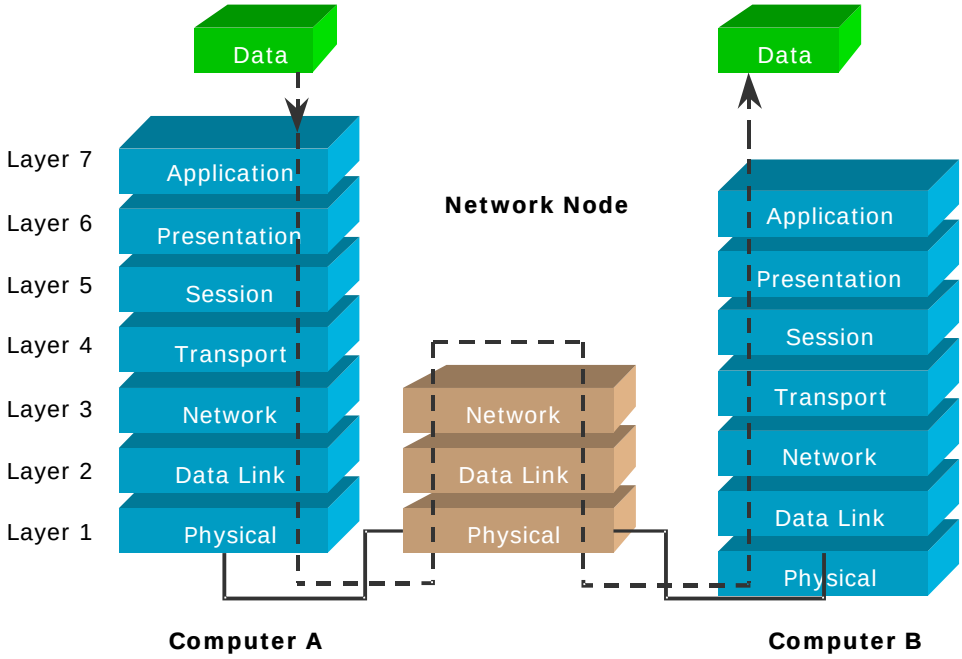


Figure 6. The OSI Reference Model

The OSI Reference model is structured into seven layers that represent the logical sequence of functions carried out when messages are constructed for transmission, dispatched, and then dismantled on arrival [Simon96]. It also serves to provide a common basis for the co-ordination of communication systems standards development and to allow existing standards to be placed into perspective [Sloman87]. An example of the OSI Reference Model is shown in Figure 6. Data at Host A is translated by the OSI stack into a form that can be communicated over the wire. It is then sent over the wire (perhaps via some network nodes), before it is reconstituted at Host B by the corresponding protocol suite, before finally being made available to the destination application.

[2] Formerly the Consultative Committee for International Telegraph and Telephones (CCITT)

Of particular interest to this thesis is Layer 7 – the Application layer. The Application layer is the highest level of abstraction defined in the OSI model and is ultimately responsible for managing the communications between applications. It provides programming primitives that a developer is able to use to access the communication facilities offered by the full protocol suite.

2.3.1 Commentary

In the previous section, we have briefly examined the emergence of communication protocols, and protocol suites, that support distributed computing. Their role and existence has been vital in ensuring we are able to successfully network our computers. In themselves, protocol suites form a hierarchy of abstractions. They provide a mechanism for translating a signal on the wire up through the layers of abstraction until at the application layer the information can be manipulated via programming primitives. These primitives bear little resemblance to their representation 'on the wire' but a developer is able to call upon the communication facilities with relative ease. The advent of the OSI model, and particularly the Internet incarnation of that model, has made communication between distributed computers much simpler. There are now a number of well-known and widely deployed communication suites in existence [Tanenbaum96].

The OSI model, and the many incarnations of protocol suites in existence are important in that they allow computers to communicate in an agreed manner. They do not address how a distributed application may be constructed. These suites are only the enabling infrastructure. Further techniques and technology are required. In the next section, we examine the emergence of distributed systems and concentrate on developments within the application layer of the OSI model.

2.4 Distributed Systems

> "A distributed system is one in which several autonomous processors and data stores supporting processes and/or databases interact in order to co-operate and achieve an overall goal. The processes co-ordinate their activities and exchange information by means of information transferred over a communications network." [Sloman87]

To understand the evolution of distributed systems, we must briefly return to examine the history of computing systems. As discussed in Chapter 1, the end of the procedural abstraction phase indicates a paradigm shift in the way software was constructed. Instead of just building monolithic standalone programs that ran in isolation, it became evident that building systems composed of smaller co-operating programs was a more effective way to construct software. Software architects began to divide their systems into discrete elements. These elements were programs in their own right, and became known as processes. A process is a running program that consists of an environment for execution and at least one thread of control [Coulouris94]. They are smaller, more manageable entities that still execute within the same computational machine, but are separately autonomous[3].

Dividing monolithic software systems into distinct processes had advantages for manageability, but meant a method was required that would allow executing processes to communicate with each other. Finding a solution to this problem became a widely researched issue with many languages gaining new facilities and programming primitives. These new facilities became known as Inter Process Communication (IPC) [Cashin80] [Fukuoka82].

2.4.1 Inter Process Communication

An early method for communication between separate processes was a unidirectional stream of bytes, known simply as a *pipe* [Coulouris94]. On a UNIX machine, for example, a pipe can be used to join the `ls` and `more` commands, e.g. '`ls –l | more`'. The output of the `ls` process is piped as input to the `more` process.

Pipes were designed as a method for linking chains of simple data-transforming programs. Initially though, they did not support networked communication, and were not able to handle large volumes of data[4] [Tanenbaum96]. A further drawback was that the pipes were bound to a specific source and target process (`ls` and `more` respectively in the above example). *Named pipes* subsequently overcame this latter limitation, allowing pipes to exist independently of any particular process.

[3] With respect to the other processes. The operating system still controls all of the processes.

[4] Local files are able to overcome this problem.

Since all interacting processes are local to each other in IPC, it is also possible to use the computer's RAM to implement a *shared memory* facility - a common region of memory addressable by all concurrent processes. Shared memory has become an important technique for use between communicating local processes. Unfortunately, there is no inherent synchronisation in this mechanism and it is easy for one process to write a value to memory for storage, and have another process overwrite it with a new value, or even erroneous data. To combat this problem, new techniques for synchronisation between processes were developed such as semaphores [Dijkstra68b], monitors [Hoare74] and sequences [Reed79].

A further communication mechanism developed was known as a *Message queue*. Message queues allow any process to write to a named queue and for any process to read from the queue. Synchronisation is inherent in the read/write operations and the message queue, which between them can support asynchronous communication between many different processes [Simon96]. Messages are distinguished by a unique identifier or message type, but are limited by being able to hold relatively small amounts of data. Table 1 lists the early IPC communication facilities, and details their advantages and disadvantages.

Method	Advantages	Disadvantages
Pipes	Simple to use; easy to chain multiple pipes;	No network support; insecure communication;
Named Pipes	Can exist unconnected to a process;	As above;
Local Files	Can handle large volumes of data; Simple to use;	Synchronisation problems; inefficient due to repeated disk access;
Shared Memory	Very fast; very efficient;	Cannot handle large volumes of data; no inherent synchronisation
Message Queuing	Inherent synchronisation; unique identifiers;	Can only hold relatively small amounts of data;

Table 1. Inter Process Communication Facilities

As the use of these facilities proliferated, it became increasingly useful to provide them as standard components of the operating system. This was normally achieved

by providing programming primitives that system builders could then employ [Coulouris94]. An early and well-known example are the IPC primitives provided in the BSD 4.x [Leffler89] versions of the UNIX [Ritchie74] operating system. These are implemented as a software layer over the underlying transport and network layers and are based on *socket pairs*, one belonging to each of a pair of communicating processes. Sockets provide a simple way of programming distributed applications using indirect message passing communication [Simon96].

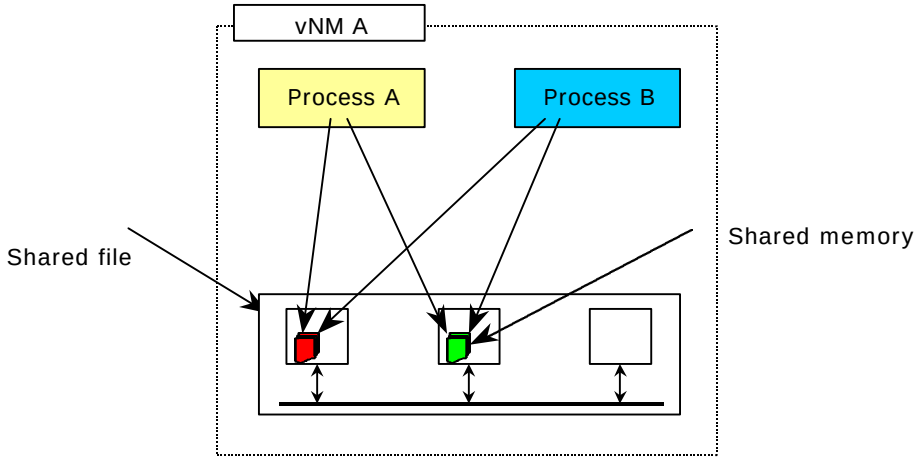


Figure 7. Inter Process Communication

In Figure 7 we see an example of IPC. Two processes are communicating by using a combination of the techniques mentioned in Table 1. By employing both local files and shared memory an optimum balance can be struck between volume of data and speed of access. Importantly, these techniques are ideal for communicating processes that exist within the same von Neumann machine.

2.4.1.1 Commentary

IPC was successful because it provided:

- simple yet effective facilities
- facilities designed for the local computing context
- facilities that were able to take advantage of local resources, e.g. memory and file space

The major factor in the success of IPC however, stemmed from the abstraction it embodies. The IPC abstraction takes full advantages of the constituent elements of the von Neumann architecture. Therefore, it is ideally suited to the underlying

hardware upon which it is used. IPC was only useful, however, for communication between processes that are executing within the same computing machine. As computer networks increased in number and size, resources were scattered even further. This distribution of resources meant that it was increasingly useful for a process on one machine to be able to access a process or resource that was located on another. Unfortunately, the existing IPC mechanisms were designed for communication between local processes only. They were complex and difficult to use in a networked manner. There was therefore a clear need for a simple mechanism to allow two networked machines to interact.

In a seminal paper, Birrel and Nelson [Birrel84] described a new mechanism, Remote Procedure Calls (RPC), which they built for the Cedar [Teitelman84] programming environment to allow remote communication.

2.4.2 Remote Procedure Calls

At their simplest, Remote Procedure Calls (RPC) are a mechanism that facilitate a request/reply interaction between two distributed processes [Simon96]. This is similar to the traditional mechanism of procedure calls [Harel93] found in high-level programming languages. The fundamental difference is that the calling procedure executes in one computing machine, and the called procedure executes in another [Cerutti93], whilst data is exchanged between the two communicating parties.

Birrel and Nelson's goal was to provide a mechanism through which remote processes could interact. They also aimed to make this mechanism *transparent* to the programmer by ensuring it was syntactically similar, and as simple for the programmer to use as ordinary procedure calls [Simon96]. Consequently, the mechanism for RPC was modelled *directly* on the IPC facilities found in the Mesa programming language [Mitchel79]. Indeed, so successful were they that RPC has no distinction in syntax between a local and a remote procedure call [Colouris94].

During an RPC call there are five separate modules that interact to enable the call. They are the client, the client-stub, the RPC communications package (RPC Runtime), the server-skeleton and the server (see Figure 8). When the client wishes to call a procedure that exists on a remote machine, it invokes the appropriate method in the client-stub. To the client, this resembles a normal local procedure call. The

client-stub then assembles one or more data packets that include the target procedure and the required arguments. These packets are then passed to the local RPC Runtime, which transmits them to the remote Runtime. On receipt, these packages are passed to the server-skeleton, where they are unpacked and passed to the target procedure in the server. Once this procedure has been executed, any results are packaged up and the process repeated in reverse. RPC is synchronous in nature, so while the server procedure is executing, the client is suspended, awaiting the result. The RPC Runtime (or request broker) establishes a client/server relationship between the interacting parties, removing the need for each party to be aware of the other's location.

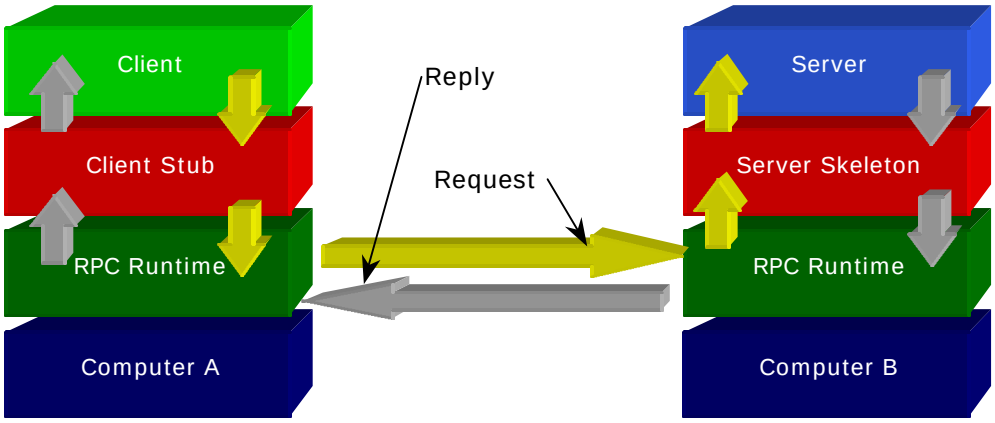


Figure 8. A Remote Procedure Call

Many RPC systems have subsequently been built, and they fall into two categories:

1] The RPC mechanism is integrated with a particular programming language that includes a notation for defining interfaces between communicating processes

2] A special purpose interface definition language that is used for describing the interfaces between clients and servers

In the first instance, languages such as Cedar, Argus [Liskov88] and Arjuna [Shrivastava89] achieve close language integration so that the requirements of remote procedure calls are handled by the language constructs themselves. The second instance includes examples such as Sun RPC [Sun89] and the Matchmaker interface language [Jones86], which have the advantage of not being tied to a specific language environment. This is achieved by having a platform neutral language that can be used to specify the names of procedures, and their required arguments, which the server is making available to the client. These specifications are known as interfaces, and are specified with an Interface Definition Language (IDL) [OMG99].

Due to its request/reply nature RPC is an extremely good way of doing Client/Server application work [Crichlow88]. Client/Server is a particular paradigm for distributing a system, where the server is a manager of one or more resources and a client is a user of that resource. The paradigm was used extensively in the 1970's to structure operating system level process interaction [Simon96] [Walsh85], and is still in extensive use today. One of the best contemporary examples being the World Wide Web [Berners-Lee92].

2.4.2.1 Commentary

The major tenets of RPC can be summarised as:

- The syntax for calling a local or remote procedure is identical
- The location of a resource is transparent to the programmer and user
- Communication is synchronous, and engenders the client/server paradigm

The early 1980's saw many breakthroughs in the distributed systems arena. Some were influenced by earlier theoretical propositions, such as communication between sequential processes [Hoare78], which were now being supported by the increasingly widespread adoption of the OSI networking suite. There were also attempts to incorporate RPC into existing programming languages, such as CONIC [Kramer83], whilst new programming languages that included distribution facilities were also developed, for example SR [Andrews82]. Again, so many proprietary and differing RPC solutions meant that the computing landscape became fractured.

In the same way that the chaos of competing, incompatible and proprietary communication protocols necessitated the creation of the OSI model, the need for a standardised model for distributed applications was recognised. In 1987, ISO began work on a Reference Model for Open Distributed Processing (RM ODP) [Brenner87] [Hutchison91] [ISO92].

2.4.3 RM-ODP

The RM-ODP model provides a framework for ODP standardisation and for the specification of systems using ODP standards [Cerutti93]. RM-ODP was an attempt to unify proprietary RPC systems, and distributed application creation. As a model, it describes in detail the application layer of the OSI model (see Figure 6). The driving

objective behind its creation was to develop a distribution infrastructure that would compliment and support the existing computing infrastructures.

Transparency Type	Proposed Advantages
Access Transparency	Enables local and remote information objects to be accessed using identical operations
Location Transparency	Enables information objects to be accessed without knowledge of their location

Table 2. Network Transparency

Like the OSI model, RM-ODP was purely a reference model. Its specification however, extends the concepts of transparency first visited by RPC, and identifies eight separate forms of transparency. These are discussed further by [Colouris94], but for the purpose of this thesis, it is suffice to demonstrate that transparency is a fundamental tenet of the RM-ODP model. We are only concerned with access and location transparency, collectively known as *network transparency* (see Table 2). Their presence or absence most strongly affects the utilisation of distributed resources [Colouris94].

Since its specification there have been a number of distributed infrastructures created that are based upon the RM-ODP model. These include the Open Software Foundation (OSF)'s Distributed Computing Environment (DCE) [OSF92], the Computer Integrated Manufacturing – Building Integrated Open SYStems framework (CIM-BIOSYS) [Gascoigne94], Sun's Remote Method Invocation (RMI) [Sun98], Microsoft's Distributed Component Object Model (DCOM) [Redmond97] and the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [OMG94]. Some of the more recent infrastructures integrate RPC with the object paradigm in an attempt to combine the benefits of the latter, in terms of modularity, with the established communication mechanism of the former [Picco98].

2.4.3.1 Commentary

In a manner similar to the process observed in Chapter 1, the abstractions that have been created to support the construction of distributed systems have gradually been layered upon each other, continually reaching ever higher.

In Figure 9 we see the evolution of distribution abstractions. IPC first came into existence as an abstraction to enable communication between processes executing within the same computer, or von Neumann machine (vNM). So successful was this abstraction that Birrel and Nelson designed RPC in an attempt to enable remote and local calls to appear identical. Out of the confusion of proprietary RPC implementations, the RM-ODP model was born, which in turn has led to contemporary distribution infrastructures such as CORBA or RMI.

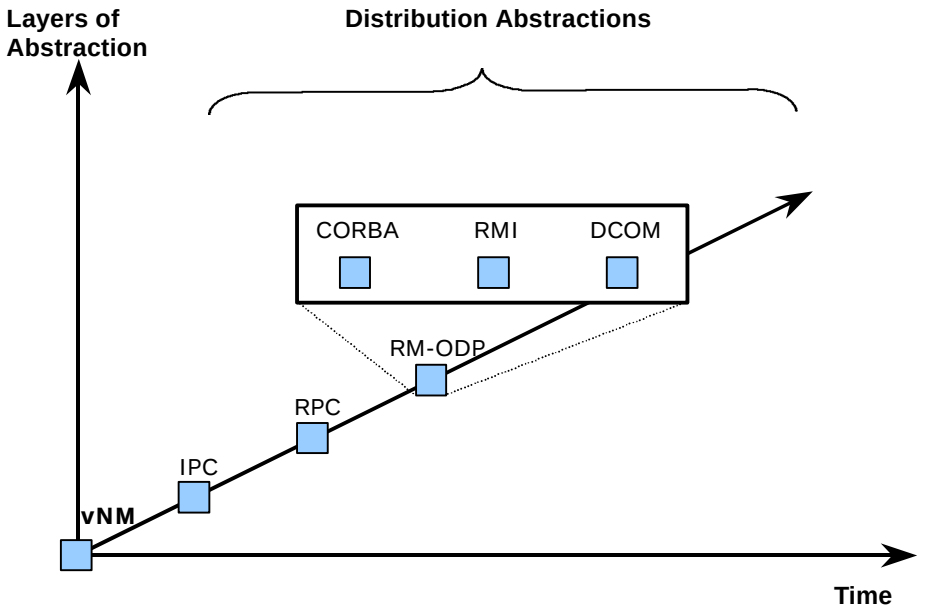


Figure 9. The evolution of Distribution Abstractions

By following the location transparency abstraction, contemporary distribution infrastructures in effect attempt to provide a virtual von Neumann machine. That is, by trying to fool every component in the system that they exist within the same address space, the overall effect is the creation of a virtual machine. Figure 10 shows an example of a distributed system built with the RM-ODP abstraction. The request broker provides a “plane of transparency” to the interacting processes.

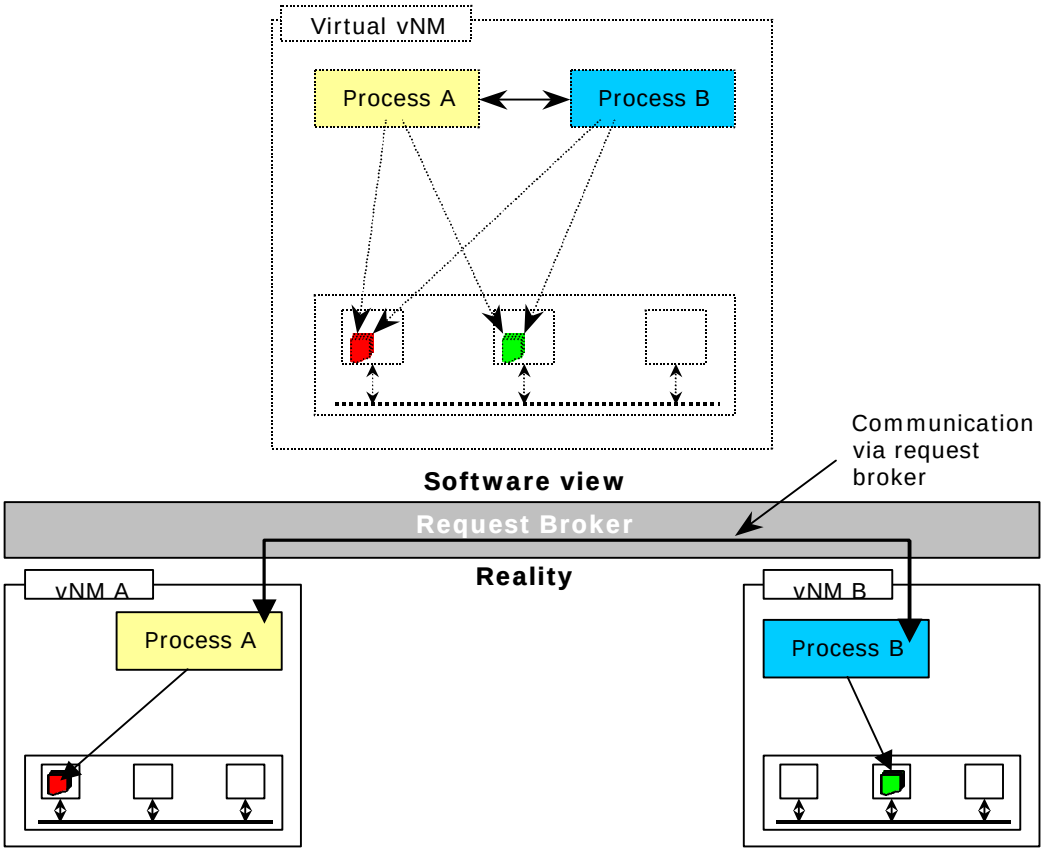


Figure 10. Request Broker providing location transparency

In reality, processes A and B exist within two complete separate vNMs, as do the resources they share. However, the infrastructure attempts to create the illusion that they exist within the same vNM. It also ensures that any required resources appear to each process as if they were in their local computing machine, thus achieving the location transparency described above.

We have now examined the emergence of contemporary abstractions and infrastructures for distribution. If we are to compare and contrast them with the Mobile Code abstraction then they must be generically categorised.

2.5 Characterisation of Traditional Distribution Architectures

So far in this chapter, we have discussed the history and emergence of contemporary distribution infrastructures. Although vendor specific (with the exception of CORBA), these infrastructures are competing implementations of the same generic type of distributed system. They share a common heritage and are each instantiations

of the RM-ODP abstraction, which itself can be traced back to RPC. For example, CORBA IDL is directly modelled on RPC.

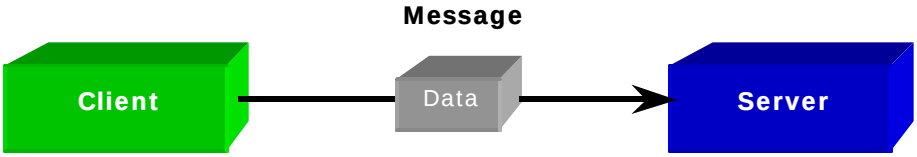


Figure 11. Mobile Data in a Traditional Distributed System

In this thesis, these systems will be characterised as distributed system infrastructures whose fundamental tenet for distribution is one of *location transparency*. They achieve this by allowing distributed systems to interact via an intermediary communications bus. The bus (or request broker) establishes a client/server relationship between the interacting parties, removing the need for each party to be aware of the other's location. The underlying communication mechanism supporting distribution will be characterised as *mobile data*.

2.6 Commentary

We have seen in Chapter 1 that modern day computing abstractions can trace their ancestry back to the original von Neumann architecture. As each abstraction has emerged, bringing with it new facilities and technologies, it has added a new layer to the continually ascending edifice. At their root though, the von Neumann architecture remains, influencing modern day designs even from the past. It is the base abstraction, the underlying model for our computational machines. As each new abstraction is layered onto the others, it must take into account those that preceded it.

When Birrel and Nelson first designed RPC in 1984, their intention was to allow the programmer to access and communicate with processes on remote machines, in the same easy manner in which they were able to access local processes. They wished to make calls to remote processes appear identical to those made locally, thereby making the location of the process transparent to the programmer (and ultimately the user). It should not matter if the process was being executed locally or on a machine on the other side of the world, it would appear exactly the same in both cases.

This phase in the development of distributed systems is pivotal. RPC was directly modelled on IPC, which had been an extremely successful mechanism for enabling

processes to communicate, and so Birrel and Nelson's intentions were not without merit. However, IPC had evolved by extending the abstractions offered by existing programming languages and by taking advantage of local facilities such as memory or file space, each fundamental constituents of the vNM. IPC therefore was a perfect abstraction for communication between processes executing in the same computational machine, i.e. in the same von Neumann machine.

RPC on the other hand attempts to mask any details of location from communicating processes. In effect, blurring the demarcation between separate vNMs to make local and remote calls look identical. The technique required to achieve this is complex; for two processes to communicate, a set of five separate modules is required (see Figure 8). Nonetheless, this technique was successful for the time, and the central tenet of the abstraction, location transparency, became one of the underlying principals for the RM-ODP model, and consequently most contemporary distribution infrastructures. Part of the reason behind the success of RPC is because it is perfectly suited to building client/server software systems. At the time, business software was predominately hosted on centralised mainframe computers, computer networks were predominately LANs or WANs and the number of personal computers was dramatically lower than today. Equally, concurrent programming was slowly becoming a reality and objects were only just gaining momentum. Thus, is it is not difficult to see why the RPC abstraction was employed successfully for the types of software system being constructed at the time. Further, it follows that such a successful technique would be used as the baseline for newer distribution infrastructures such as CORBA. These new infrastructures take this issue further, creating what in effect is a virtual vNM, where the illusion is created that all components in the system exist within the same computational machine (see Figure 10).

Since that time, the nature of the environment in which these distributed systems exist has been changing. Fuelled by the Microsoft vision of a PC on every desk, personal computers have taken over many of the responsibilities that used to be the domain of the mainframe. The network has also seen a dramatic enlargement with the explosion of the Internet, but has also suffered from quality of service issues. Object-oriented programming has fundamentally changed the way we view software systems, moving

us away from the synchronous single threaded model, to one that includes asynchrony, multi-threading, encapsulation and component reuse. In short, many of the assumptions made in the creation of RPC have now become erroneous. For example, RPC implicitly assumes that the network is 100% reliable, and thus that remote procedures will always be available. Anyone who has used the Internet will attest this as a fallacy.

By 1994, the first strong doubts over the validity of the RPC approach were being raised. In a seminal paper, Waldo *et al* [Waldo94] argue that objects[5] acting in a distributed system are intrinsically different to those in a local system and therefore must be treated very differently. They identify four major problem areas when comparing local and distributed systems (see Table 3).

Problem	Details
Latency	• Can be up to a difference of 4-5 orders of magnitude • Most obvious • Least worrisome
Memory Access	• Unable to use pointers • Because memory is both local and remote, call types have to differ • No possibility of shared memory
Partial Failure	• Is a defining problem of distributed computing • Not possible in local computing
Concurrency	• Adds significant overhead to programming model • No programmer control of method invocation order

Table 3. Problems of a Distributed System

In particular, partial failure is identified as an extreme problem for distributed computing. Sloman had earlier expressed the view that:

> "If the programmer is to take advantage of location transparency, this means that the behaviour must be the same in both cases [local and remote]. This can be costly and difficult to achieve, especially in the face of failures" [Sloman87]

[5] This applies equally to processes and procedures, etc

In addition, even before the Waldo paper, Nelson himself had suggested that:

> "If the aim is to provide location transparency then we must aim to provide the same behaviour as in the case of a failure in a local procedure call, although this can be costly." [Nelson81]

In Figure 12, we see a software system built with the RM-ODP abstraction distributed over three vNMs. Each component has access to certain resources, but of course, there is no way for the component to tell if the resource is local (within the same vNM) or remote. In the case of remote resources, the request broker is required to support the illusion that they are indeed local, by providing the relevant connections "behind the scenes". This is depicted by the lines flowing through the plane of transparency. From this very simple hypothetical system, it is evident just how many lines cross the boundaries of vNMs. At each crossing, the system is subject to the types of problem identified in Table 3.

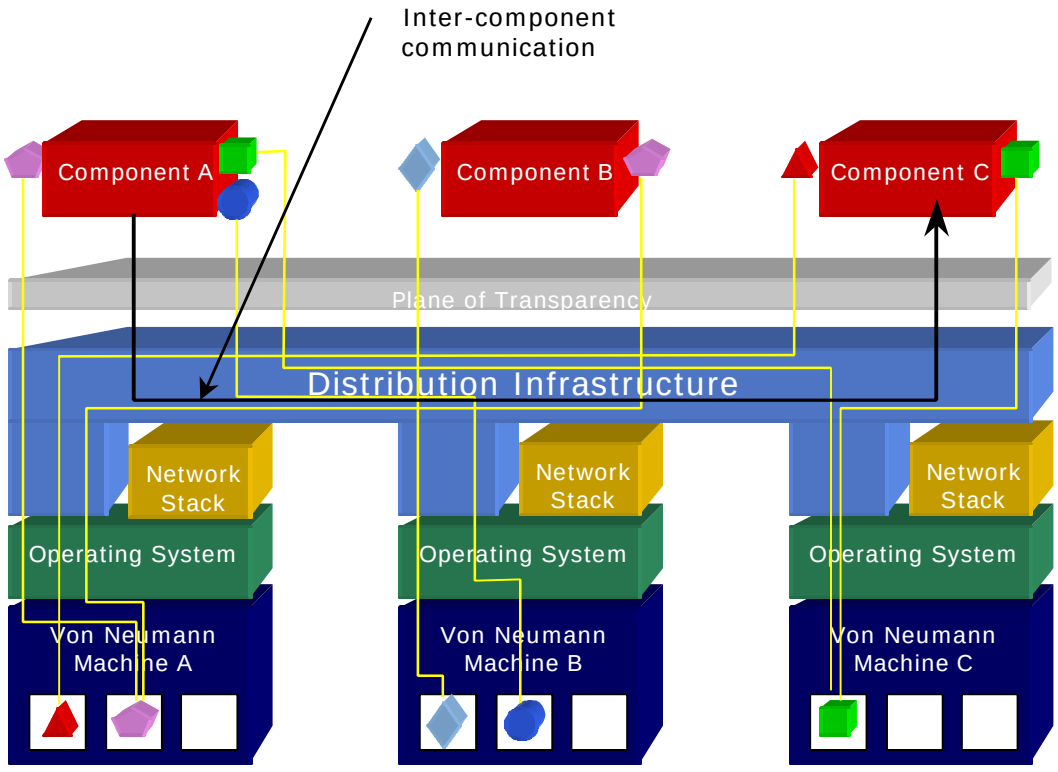


Figure 12. Back flips required by ORB to ensure location transparency

The central thesis of the Waldo paper is that local and remote computing are just plain different, and should be treated as such. They argue that distributed systems should be built with the premise that there are two distinct types of objects: local objects and

remote objects. Although Waldo *et al* identify the key differences between local and distributed computing, their discussion of why these make distributed computing different are pragmatic. The differences are eloquently stated, but there is no reason given for exactly *why* these differences are evident, just that they are – and that the two types of computing should be treated differently. In this part of the thesis, we go further and present an argument as to the cause of these differences.

We have seen that IPC was an ideal abstraction for interacting processes within the same vNM. Its success was built on the fundamental elements of a vNM, i.e. a single memory (that could be shared), a single CPU and local files (I/O). RPC attempts to take this effective abstraction and make it apply to many vNMs, by making location transparent. This is similar to many contemporary distribution infrastructures. Indeed, the stated goal of the Millennium experiment undertaken at Microsoft Research is:

> "… to eliminate completely the distinction between distributed and local computing … by raising the level of abstraction so that programmers are not even aware of distribution" [MSR98]

However, practice has shown that this approach is fraught with difficulties [Waldo94], and the discontinuation of this project serves as a clear indication.

Certainly then, there are two diametric views as to how we may build reliable distributed systems.

1] Use an abstraction that completely removes any knowledge of location
2] Use an abstraction that views remote and local objects as completely different

This thesis supports the assertions of Waldo *et al*, i.e. that we should treat local and remote objects differently. However, we go further and argue that the fundamental reason that RPC, and thus contemporary distributed systems based on the RM-ODP abstraction, suffer from the problem mentioned above is because of the underlying abstraction they embody. The RPC abstraction pays little regard to the supporting layers beneath it; rather it attempts to strike out on a new course of its own and is unsuitable for the underlying hardware substrate. Instead of continuing the long line of abstractions that have served so well, RPC attempts to impose an abstraction that is perfect for one vNM onto many. It pays little attention to the underlying hardware

abstraction, which as we have seen is the vNM. RPC has broken the abstraction tower, and it is this fact that causes the acute problems associated with distributed systems that Waldo et al have identified. While the RPC approach has been, and continues to be, useful under certain circumstances, it no longer supports the type of distributed system we wish to build in today's networks with current software engineering techniques and technologies.

2.7 Concluding Remarks

> "It can be argued that RPCs should not be entirely transparent as their semantics and performance differ from those of local procedure calls." [Colouris94]

> "… a number of distributed systems have attempted to paper over the distinction between local and remote objects [and failed]. These failures have been masked in the past by the small size of the systems." [Waldo94]

As computers have become more prevalent, and the resources they represent the lifeblood of business, we have developed methods for connecting computers and enabling them to communicate with each other. Once communication was achieved it was only natural that we pursue techniques for building software systems that span multiple hosts, allowing us to harness the additional power and multiple resources made available.

In this chapter, we have examined the emergence of distribution, and traced the evolution of abstractions used to build networks. Networks are an essential constituent of distribution, they enable communication between computers. They are the substrate over which distributed systems can be built. Next, we have examined the evolution of abstractions used in contemporary distributed systems. We have seen how RPC attempts to extend the extremely successful IPC abstraction, ultimately leading to the *location transparency* abstraction, embodied in many contemporary distributed infrastructures. In effect, these infrastructures attempt to create a virtual von Neumann machine. This approach has been shown to be unreliable.

The central thesis in this chapter is that by attempting to create the illusion that all components exist within the same machine, location transparency is breaking the

layers of abstractions upon which computing has been built since the dawn of computing. The abstraction is unsuitable for the underlying computational machine upon which it must execute. We need new techniques and abstractions for distributed computing that do not break our layers of abstraction, rather they continue to appreciate what has preceded them, and are suited to the underlying computational machine. In the next chapter, we review mobile code, a new technology that promises to fulfil these requirements.

3 Mobility

3.1 Introduction

Code mobility is not a completely new idea. There have been several widely used and successful mechanisms for moving code around a network previously employed, perhaps the best known being the PostScript language [Adobe85] that is used to control printers.

Recently though, mobility has been examined from a different perspective, and has become a burgeoning topic for discussion in mainstream distributed systems research. Mobility currently boasts a flourishing research community dedicated to investigating the potential of this new paradigm [Mobility99]. So far in this thesis, we have built an argument against using location transparency, the abstraction embodied in contemporary distributed systems. We have identified the need for new abstractions for distribution, which are entirely suited to the underlying computational machine, and are able to distinguish between local and remote resources.

In this chapter, we conclude Part I of the thesis, the philosophical argument concerning the abstractions employed in building distributed systems. We begin by reviewing mobile code abstractions and examining the differences between systems built with these abstractions and contemporary distributed systems. Finally, we discuss what makes mobile code systems different, and why the abstractions they embody are more suited to distribution than location transparency.

3.2 A Brief History of Code Mobility

There have been previous examples of code mobility. One of the earliest being remote batch job submission [Boggs73]. Employed at the time of hugely expensive central mainframes, batch job submission allowed users to submit code for execution on the server. Although working at a very basic level, this technique was a mainstay of computing life when both processor time and core resources were scarce. In effect, batch job submission allowed computation to be moved from one location to another to take advantage of local resources, although the movement required manual intervention by the user.

This basic concept was the seed for further research, and out of it grew projects such as Accent [Rashid81] and RIG [Rashid86], which culminated in the MACH [Accetta86] operating system. These were experiments in building distributed operating systems, which attempted to present the same abstractions regardless of the underlying hardware substrate. Latterly, this work has been embodied in migratory systems such as Locus [Thiel91] and Cool [Lea93], which support process and object migration respectively. Both systems provide mobility at the operating system level, and therefore any migration is transparent to the user and system programmer. As argued in Chapter 2 though, complete transparency can be counter-productive. Certainly, the designers of Emerald [Jul88] concur, as they offer the programmer explicit control over migration, as well as automatic migration.

Thus far, the techniques described have been positioned at the operating system level and are particularly useful when dealing with small scale distributed systems. They do not tend to be suitable for large-scale networks and systems, particularly those of the scale of the Internet, and have mainly been used for techniques such as load balancing [Picco98]. Although process migration never took off as a commercial reality, the research was widely regarded as successful [Milojicic99].

The notion of mobile computation at a higher level of abstraction was first suggested in “Objectworld” [Tsichritzis85], a hypothetical computing environment geared towards information dissemination in which all objects could be mobile. This, and the ideas embodied in migratory systems have spawned a new field of research that is investigating similar solutions but on a much larger scale and at a higher level of abstraction. This field has many names, amongst them mobile code systems, mobile object systems, active networks and mobile agents. For the remainder of this thesis, we use the terms interchangeably unless explicitly stated otherwise. Unfortunately, there is still no consensus among the mobility research community as to what exactly each term refers to, or a standard definition for each to which everyone subscribes. Therefore, in this thesis we define a mobile agent as:

> “a software agent that is able to autonomously migrate from one host to another in a computer network.” [Papaioannou99]

The notion of a mobile agent was first established in 1994 with the release of a white paper by White [White94] that described a computational environment known as "Telescript" [White96]. In this environment, executing programs were able to transport themselves from one node to another in a computer network, in order to interact locally with resources at those nodes. Telescript was never a commercial success, but it did generate a lot of academic interest.

Since that time, this field has exploded in popularity, with a plethora of new frameworks and infrastructures appearing almost continually [MAL99]. This profusion of experimental frameworks is reminiscent of the explosion of new programming languages in the early days of computing (see Chapter 1) and is indicative of a new and immature research field. Although we review some of the more popular mobile code systems in the next chapter, to fully understand this new paradigm we must first examine the differences between contemporary and mobile code based distributed systems.

3.3 The Differences

In Chapter 2, we saw that the central tenet and abstraction of contemporary distributed systems is *location transparency*, with inter-component communication being achieved via an intermediary communications broker. For both the programmer and the system components, this abstraction provides no notion of location. Instead, the *distribution infrastructure* enforces a "plane of transparency" in an attempt to create a virtual computational machine above the network layer. The abstraction hides any details of the underlying hardware, and attempts to create the illusion that every component of a distributed system exists within the same computational machine. Unfortunately, this approach is subject to the many problems identified by Waldo *et al* (see Section 2.6). This thesis argues that the location transparency abstraction is fundamentally flawed, as it breaks the Tower of Abstractions by attempting to impose an unsuitable abstraction on the underlying computational substrate.

Distributed systems built around the tenet of mobile code are quite different. Instead of masking the physical location of a component, mobile code infrastructures make it evident. These systems embody a completely different abstraction. Each node in the network has an *Executing Environment* (EE) through which components are able to

access the facilities of the network layer. These facilities can then be used to communicate with other remote components as normal. However, if components require access to a resource that is not located at their current host, or wish to interact locally with another component, they are able to migrate to the new host. In Figure 13, we see examples of the mobile code paradigm. Component A is in communication with Component B, both of which have references to local resources. However, in contrast to contemporary distributed systems, A requires explicit knowledge of the location of B so that they may communicate. There is no request broker to mediate the communication. Component C is separate, and demonstrates the mobility aspect of this approach. Instead of communicating with a data source across the network, C is able to migrate to the data source's host, and interact with it locally. In a contemporary system, C would not even be aware that the data source resided on a different host.

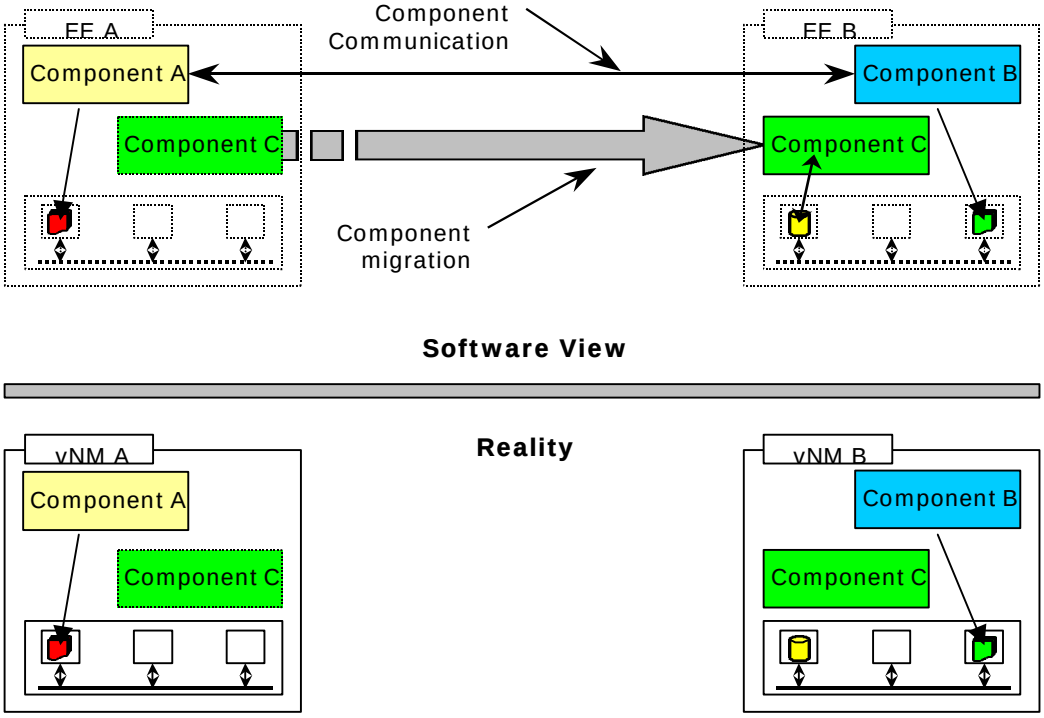


Figure 13. Communcation across the network, and mobile agent migration.

The major differences between mobile and contemporary distributed systems are well described by Picco [Picco98] and are summarized here:

- **Code mobility is geared for Internet-scale systems –** systems such as Emerald and Locus were designed with small-scale networks in mind. Thus, they assume high

bandwidth, reliable networks, small latency, trust, and homogeneity. Mobile agents on the other hand are built with the opposite criteria in mind.

- **Programming is location aware –** mobile agent systems provide an abstraction in which the notion of location is available to the programmer and the constituent components of the system.
- **Mobility is a choice –** migration is controlled by the programmer or at runtime by the agent, instead of being triggered transparently by the system.
- **Load balancing is not the driving force** - process and object migration operating systems were primarily designed to assist with resource and load balancing. Mobile agents are used to design systems supporting flexibility, autonomy and disconnected operation.

Mobile code is a powerful programming abstraction offering many possibilities. To fully appreciate and employ successfully, it is important to understand all the nuances of the different architectural abstractions afforded to the system designer. In the following sections, we describe the different flavours of the mobile code paradigm.

3.4 Mobile Code Design Abstractions

To discuss differences in design abstraction we require a context in which to examine each abstraction. Further, we must define common concepts that may be used to perform our analysis. In the following examples, *Components* are the constituent parts of a software system. They execute within an execution environment at a particular *Host*. Components may contain *Logic*, an encapsulation of the knowledge required to perform a certain *Task*. Completion of this task may also require access to a *Resource*. Components may interact with each other via *Message* passing, in which each message may contain pure data, logic or both. In addition, components are able to migrate to a new host if they so desire. Examples of each abstraction are shown in Figure 14.

3.4.1 Remote Computation

In remote computation, components in the system are static, whereas logic can be mobile. For example, component A, at Host H_A, contains the required logic L to perform a particular task T, but does not have access to the required resources R to complete the task. R can be found at H_B, so A forwards the logic to component B, which also resides at H_B. B then executes the logic before returning the result to A. This is how the aforementioned remote batch entries [Boggs73] work.

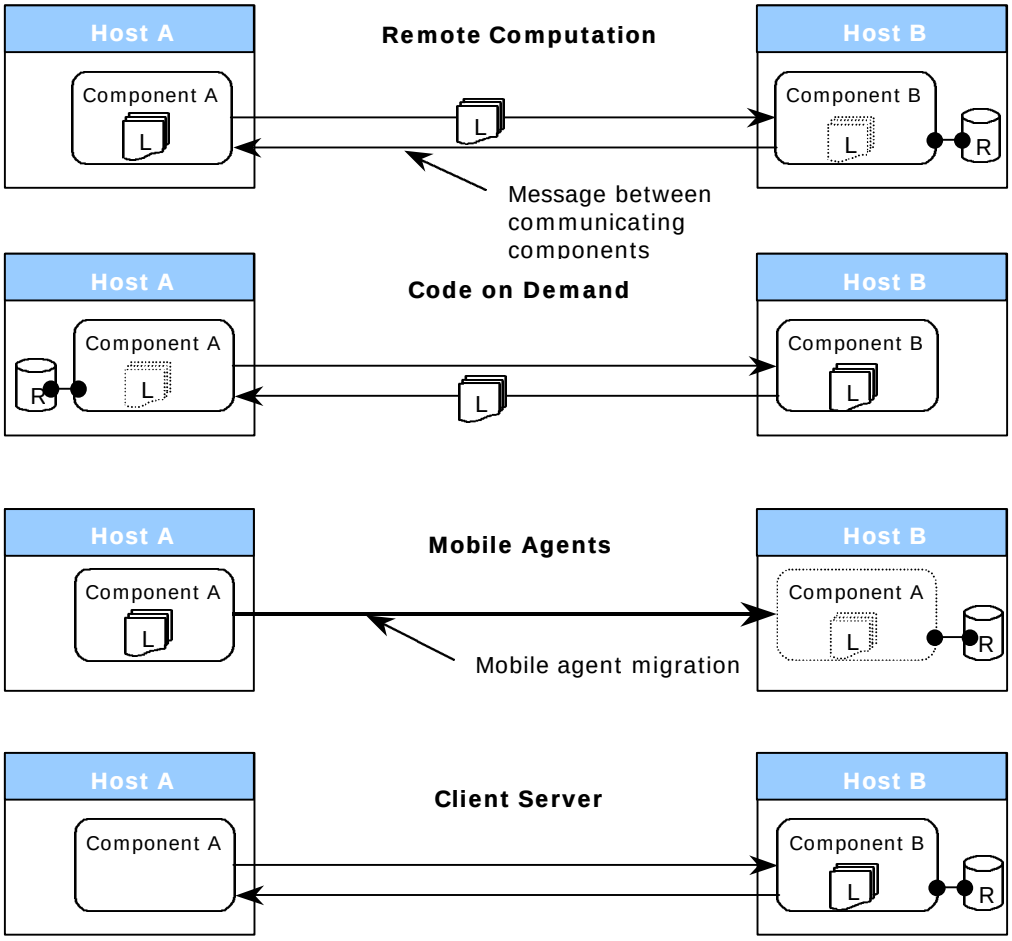


Figure 14. Examples of the different mobile code abstractions.

3.4.2 Code on Demand

In Code on Demand, component A already has access to resource R. However, A (or any other components at Host A) has no idea of the logic required to perform task T. Thus, A sends a request to B for it to forward the logic L. Upon receipt, A is then able to perform T. An example of this abstraction is a Java applet, in which a piece of code is downloaded from a web server by a web browser and then executed.

3.4.3 Mobile Agents

With the mobile agent paradigm, component A already has the logic L required to perform task T, but again does not have access to resource R. This resource can be found at H_B. This time however, instead of forwarding/requesting L to/from another component, component A itself is able to migrate to the new host and interact locally

with R to perform T. This method is quite different to the previous two examples, in this instance an entire component is migrating, along with its associated data and logic. This is potentially the most interesting example of all the mobile code abstractions. There are currently no contemporary examples of this approach, but we examine its capabilities in the next section.

3.4.4 Client/Server

Client/Server is a well known architectural abstraction that has been employed since the first computers began to communicate. In this example, B has the logic L to carry out Task T, and has access to resource R. Component A has none of these, and is unable to transport itself. Therefore, for A to obtain the result of T, it must resort to sending a request to B, prompting B to carry out Task T. The result is then communicated back to A when completed.

3.4.5 Subtleties of the Mobile Agent abstraction

Although all of the mobile code abstractions are ostensibly similar, there are some fundamental differences, which have substantial implications for which particular abstraction to employ. In this section, we highlight one of the key issues that differentiate the abstractions, multi-hop mobility. Multi-hop mobility refers to the ability of a mobile agent to migrate to more than one host, taking action at successive hosts in order to fulfill some goals. The destination of the next host may only be determined at the present host, and does not have to be known at the outset of the journey. In contrast, the other mobile code abstractions are utilized at best as mobile messengers, that do not continue to further hosts once they have performed their tasks, or at worst as techniques for shipping code around a network. For example, let us hypothesize a situation where a BookAgent has queried all StoreFrontAgents and is unable to fulfil its Order. It then has to contact the WarehouseAgent to ask whether a copy can be allocated from there, or when the next copy will arrive. In a contemporary client/server architecture, this would require many calls to remote processes before the task had been complete. Each time a call is made across the network the system runs the risk of the Waldo problems. On the other hand, a mobile agent is able to migrate from host to host, and interact with the StoreFrontAgents locally, before finally arriving at the host of the WarehouseAgent. Once there, it can

begin a new dialogue with the WarehouseAgent to establish when the required book will become available. This scenario is depicted in Figure 15 below.

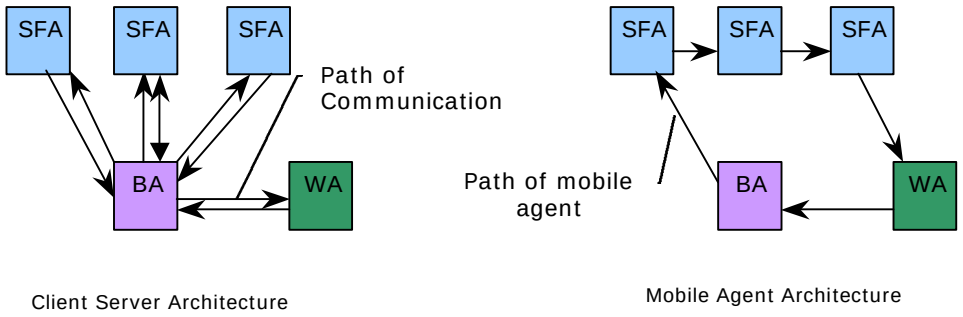


Figure 15. Network routing of Client/Server and Mobile Agent architectures

From these diagrams, it is evident that a mobile agent architecture involves less recourse to network communication than a client/server architecture in this particular scenario. In addition, each time the mobile agent is using the network it is to transport itself, not make a remote call to a component on another machine. If we imagine that each interaction entailed more than a simple request/reply dialogue then the client/server diagram would quickly become littered with communication arrows, whilst the mobile agent one would remain identical. The ability to move the computation to the data source and continue locally is one of the biggest advantages of mobile agents.

3.5 Characterisation of Mobile Agent Systems

Although we have examined several abstractions that are part of the mobile code family, the one with the greatest potential is undoubtedly the mobile agent abstraction. In this thesis, mobile agent systems will be characterised as enabling distributed systems by supporting *local interaction* and mobile *logic and data*.

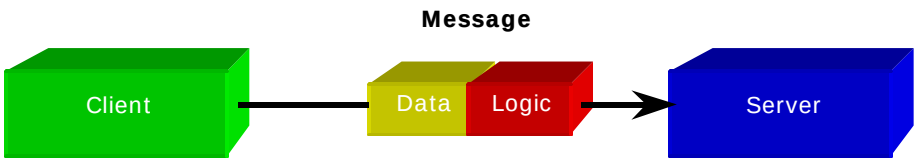


Figure 16. Mobile logic *and* data in the Mobile Agent Abstraction

This is very different to the characterisation in Section 2.5 of the messaging in a distributed system built with the location transparency abstraction.

3.6 Commentary

In Chapter 1, we traced the evolution of computing from the early work of von Neumann through to the present day. We followed the emergence of computing abstractions, and saw how those we employ have been gradually layered upon each other, forming a continually ascending tower of abstractions, whilst retaining as their underlying computational model and base abstraction the von Neumann machine.

In Chapter 2, we examined the emergence of distribution. We saw how RPC attempts to extend the successful abstraction of IPC onto many computational machines by promoting *location transparency*, an abstraction that would manifest itself in distributed systems built around the tenets of RM-ODP. Ultimately, distributed systems built with this abstraction suffer from several major problems (see Table 3). We have argued and demonstrated that this is due to the location transparency abstraction breaking the Tower of Abstractions that has been built to enable and support computing. In short, we argue that location transparency is an unsuitable abstraction for distribution for the underlying computational model.

In this chapter, we have reviewed a new paradigm, with new abstractions, that potentially fulfils the requirements for a distribution abstraction put forward earlier in Chapter 2. Our requirements may be summarised as follows.

A distribution abstraction:

- that remains faithful to the underlying von Neumann machine
- that does not break the tower of abstractions
- that is able to differentiate between local and remote components

It is precisely these requirements that the mobile code paradigm fulfils. As we have seen, its central tenet is one of *local interaction*. Components in a distributed system that wish to communicate are able to transport themselves across the network so they may interact locally at the same host. In addition, components are also able to communicate by exchanging messages across the network.

In each case, the core abstraction remains faithful to the underlying von Neumann machine and the Tower of Abstractions. Instead of attempting to remove location from the abstraction, and build a virtual computational machine, mobile code makes

location evident. It is a central aspect of the abstraction, and enables designers to make a judgement on how components might communicate. Indeed, the *execution environment* of a mobile code system may itself be viewed as an additional virtual computational machine being added to the Tower, but it remains consistent with the underlying base abstraction. By ensuring that any protracted communication is done locally, components are able to return to the successes of IPC by taking advantage of the core facilities of the vNM, e.g. shared memory and files. Instead of attempting to achieve distribution by imposing an unsuitable abstraction across many machines, mobile code simply layers a new abstraction upon the existing tower; a time honoured route to success. In fact, we argue that local interaction as embodied in mobile code systems should be viewed as a successful adaptation of IPC to distribution.

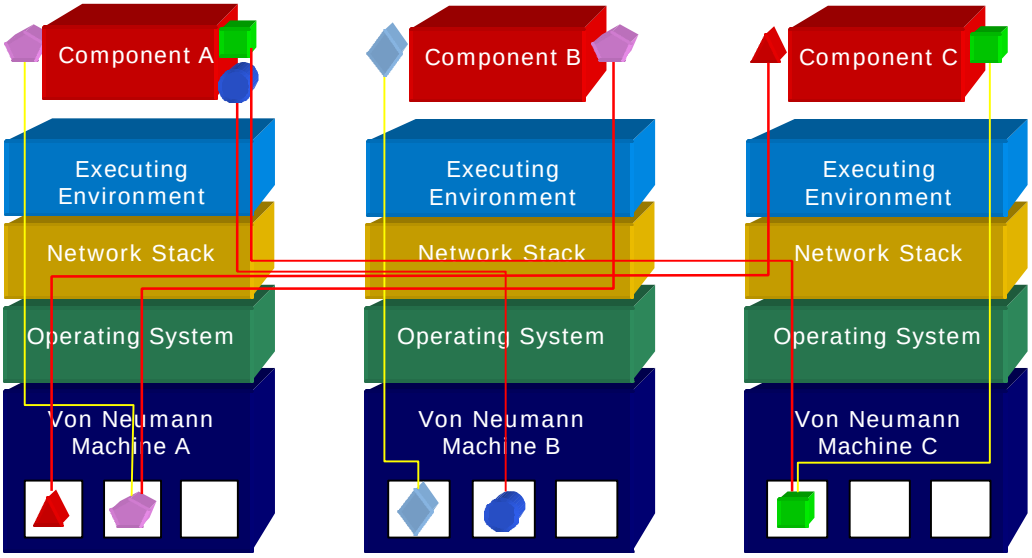


Figure 17. A distributed system built with mobile code

In Figure 17, we see the same hypothetical distributed system that was first encountered in Chapter 2. However, this time the system has been built with the mobile code paradigm. Again, each process has access to certain resources, but this time there is clear knowledge of the location of each resource, i.e. in which vNM it resides. Local references are shown in yellow, whilst remote references are shown in red. Knowledge of the location of a resource, allows each component to make a judgement about the type of reference it holds to that resource. In comparison to the RM-ODP version of this model, there is no illusion being created by the “plane of transparency”. While network references may still suffer from the problems depicted in Table 3, the components themselves are aware that this is a potential problem. In

addition, if a component decides it would be beneficial to be located at the same host as a resource it may migrate to take advantage of local interaction. For example, in the case of component C, when it has finished interacting with the green cube, it may migrate to vNM A to communicate locally with the red triangle.

The major conceptual difference between the two distribution abstractions is clear, *location*. With location transparency, location is removed from the abstraction and a virtual computational machine is created which attempts to create the illusion that all components in a system reside within the same address space. The illusion, however, can be shattered by any number of problems associated with trying to create a rock solid abstraction across the network.

In contrast, local interaction makes location evident and components are able to make a judgement themselves about how to communicate with other components. It is this fundamental difference that the author believes is vitally important. In Chapter 1, we discussed how abstraction is an immensely powerful tool. It allows us to manage the complexity of a situation, and to rationalise about it by removing those details we consider inessential. It is the author's belief that when it comes to distribution, location is a vital piece of information. We are no longer attempting to build distributed systems in networks in which location can be papered over, in which the size of the system can mask the fallacies in the paradigm. We are now building large systems in which the network is unreliable, in which the topology of the network or availability of resources may change rapidly. In such an environment, information about location becomes essential. If we examine perhaps the most successful distributed system of all time, the Internet, we see that location is central to its success. The URL [Berners-Lee92b] abstraction is purely a reference to location, but has been fundamental to the evolution and success of the web. We must learn from these lessons.

3.7 Concluding Remarks

> "Keep design as simple as possible, but no simpler" [Einstein39]
>
> "A designer knows that he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away" [Antoine de Saint-Exupery]

We have seen throughout Part I of this thesis how important abstraction is to computing. It is the central essence of an idea or design. Abstractions allow us to remove the details and focus on the essence of a situation. Any specific example of a technology is merely an instantiation of the abstraction. The majority of the history and evolution of computing has been concentrated on the development of new abstractions. Our current abstractions for distribution have proved limiting and unreliable. We require new abstractions to support distributed computing on a hitherto unforeseen scale. Mobile Code systems are one such solution.

In Part I of this thesis we have built a philosophical argument concerning the abstractions used in building distributed systems. It is our belief that the location transparency abstraction, as embodied in the RM-ODP model, is fundamentally unsuited to the underlying hardware substrate. Instead of attempting to utilise the strengths of preceding abstractions, location transparency enforces a "plane of transparency" whose purpose is to create the illusion of co-location and to mask any details of distribution from components in the system. The abstraction views location as a detail that can be removed.

Local interaction on the other hand remains faithful to the core abstraction, and makes use of the core facilities embodied in IPC. Instead of masking location, it makes it evident. Communicating components are aware if they are local or remote to each other, and are able to make a judgement about how to communicate. By utilising the strengths of the von Neumann machine *and* the network, the local interaction abstraction allows us to build distributed systems that do not suffer from the Waldo [Waldo94] problems.

The central argument of Part I is that local interaction should be the abstraction of choice for building distributed systems. In hindsight, we should view location transparency as an evolutionary blip, a wrong fork in the road. If we are to build successful distributed systems in the myriad of new networks, we must be bold and admit our mistakes of the past.

Part II

Using and Evaluating

4 Mobility in the Real World

4.1 Introduction

Mobile Code is a new and generally untested paradigm for building distributed systems. Although garnering many plaudits and continually increasing in popularity, the technology and research field remain relatively immature [Picco98]. To date, most research has focused on the creation of mobile code frameworks, and as yet there is no consensus on a conceptual framework with which to compare results. Further, there is no clear understanding of the new abstractions offered by this paradigm. Part I of this thesis aspires to address the conceptual deficiencies of the research field by offering a philosophical argument and critique of mobility.

In Part II we begin our study of mobility in the real world. In later sections of the chapter, we will see that there are many advantages claimed for mobile code systems. Unfortunately, these claims remain qualitative and subjective in their nature. The dearth of quantitative results, however, means it has not yet been possible to properly evaluate the potential of either the technology or the paradigm. In the last year a trickle of results is beginning to validate some of the claims [Papastavrou99] [Picco98b], and these results are certainly important in establishing the credibility of mobile code systems. Nonetheless, it is the author's belief that these types of improvement are optimisations, or incremental improvements. The true benefit of the paradigm is in the type of software architecture that can be built. In support of our arguments presented in Part I, in Part II we provide an insight into how well mobile code architectures respond to real world pressures.

4.2 Research Motivation

In Part I, *Understanding*, we have presented an argument built around a philosophical understanding and critique of the abstractions used to build distributed software systems. The central thesis is that contemporary distributed systems built with the location transparency abstraction are fundamentally flawed and that we require new abstractions for distribution. Our proposal is that a new abstraction, local interaction, is better suited to the underlying hardware substrate upon which distributed systems are built. To demonstrate this we have traced the emergence and evolution of

computing, and the abstractions that exist in this field, beginning with the early pioneering work of John von Neumann. We believe that Part I contributes to raising the level of conceptual understanding surrounding the mobile code paradigm, especially when examined in the wider context of the different abstractions embodied by distributed systems.

Although we believe the essence of any technology is the core abstraction it embodies, we understand that pure academic reasoning is never sufficient to make a valid judgement about a new technique or technology. What is required is first hand experience. Therefore, in addition to our philosophical argument, we aim to support these arguments by investigating the application of mobile code in the real world. We wish to demonstrate the feasibility of actually building distributed systems with this technology. Certainly, the arguments presented in Part I are extensive, and a full experimental investigation is beyond the scope and timescale of a PhD[6]. Instead, we must shorten our horizons and take the first steps along the long path of validation. Part II is therefore a report on our experiences of *Using and Evaluating* mobile code in the real world.

As we have seen, the technology base in the field of mobile code remains immature. Whilst the plethora of new frameworks continues to increase, the amount of real distributed systems built with this technology remains low [Milojicic99]. Although abstractions are the central essence of a paradigm, the technological instantiation of that abstraction must successfully embody it. To support our argument of Part I, we must prove that mobile code can be used to build real world systems. Thus, our research motivation is to investigate and use mobile code, as it would be in the real world, and to analyse the issues involved and the lessons that can be learnt.

In Chapter 3, we described the choice of design abstractions available to the system architect who wishes to employ mobile code. These were Remote Computation, Code on Demand, Mobile Agents and Client/Server. Since many examples of Code on Demand currently exist [Hopson96], and Client/Server architectures are an extremely well known approach, we feel these abstractions are of less interest to this study. Therefore, the implementation described in this thesis will encompass prototype

[6] Indeed, an entire academic career could be pursued with these arguments!

systems of the Remote Computation and Mobile Agent abstractions. We have gained an understanding of each abstraction, and have been able to compare the two. For ease of use, and because of the conceptual abstraction they support, from herein we refer to the former as the Mobile Object system, and the latter as the Mobile Agent system.

4.2.1 Research Objectives

As the software systems that underpin industry have become ever more complex and interlinked, the inherent flexibility of the underlying software designs has been compromised. On the small scale and under the right circumstances software systems can be extremely responsive, flexible and easy to change, for example the existence of the requisite skills. Therefore, matching a change in business practice should not be a problem. However, when examined in the large this is not the case. As observed by Cox:

> "There was a time when the virtue of software over physical media like paper and pencil was in its very responsiveness … Although this may be to some extent true for small projects (program building), it is not (and has never been) true for ambitious undertakings (system building). In fact, software systems are usually the least responsive element in many organisations today. The organisation as a whole is able to adapt more fluidly than the software upon which it has grown dependent." [Cox87]

Recent experience has shown that attempts to create large scale supporting infrastructures have resulted in complex monolithic systems that are the least flexible element within an enterprise [Barber98]. Most companies require a change in their software at some point, and so software change is one of the most important issues currently facing the software industry [Booch94]. A software system will have a limited lifetime if it cannot be altered to accommodate a change in the business process it is intended to support.

This issue is well known to the software engineering community, and in this thesis we refer to it as System Agility. There already exists a substantial body of work relating to the issue of system agility, e.g. [ICSE'99], and the full variety of issues is vast. We

cannot hope to consider them all in our experimental study, so we initially select two broad but vitally important factors on which to focus:

1] How easy the system is to understand
2] How easy it is to modify

These are still broad issues, with many factors contributing to each, so we refine our focus even further. To represent each facet, we have selected the specific issues of Semantic Alignment (SA) and Component Coupling (CC). System integration and agility has been one of the main issues of research at the MSI Research Institute for nearly a decade, and therefore SA and CC augment the research undertaken by other members of the institute [MSI99] [Coutts98b]. In the next sections, we briefly review both concepts.

4.2.2 Semantic Alignment

The ability to communicate ideas clearly and effectively was a concern for the human race even before written records began [Pinker95]. Whenever two people talk, they have only an approximate understanding of each other. When they speak the same language, share intellectual assumptions, and have common backgrounds and training, the alignment may be closer. As these factors diverge, there is an increasing need to put effort into constant calibration and readjustment of interpretations, since ordinary language freezes meanings into words and phrases, which then can be "misinterpreted" (or at least differently interpreted). Clear communication requires a shared understanding of the meaning of terms; and this understanding is known as Semantic Alignment [Clark96]. While this term has its roots in linguistics, it is also applied to software engineering. For example, if information is being shared between two company databases that have a table for "employee," they are apparently in alignment. However, if one was created for facilities planning and the other for tax accounting, they may not agree on the status of part-time, off-site, on-site contract, or other such “employees.”

A software system is invariably built to support a business process. Therefore, in the context of system agility we define Semantic Alignment as:

> "Semantic Alignment refers to how successfully a software system embodies the real world business process is it intended to support, i.e. how well the software models the real world."

For example, if in the real world a business process contained the concepts of Apples, Oranges, Potatoes and Tomatoes, but the software model only contained the concept of Food, then this system would not be as successfully aligned as a system that contained the concepts of Fruit and Vegetables.

4.2.3 Component Coupling

Component Coupling was first defined in the 1970's by Constantine and Yourdon [Yourdon79]. It is a technique for measuring the inherent maintainability and adaptability of a software system, both of which are important issues that directly affect the overall agility of a software system. In short, component coupling measures the dependencies between two software components, i.e. how many times a component depends on the functionality of another object to perform its role. It is considered desirable to limit the number of inter-object dependencies in a system, since this not only affords greater flexibility to the designer during construction, but also ensures the system remains easy to change in the future. Therefore, the objective of a designer is to limit these dependencies, thus making the system "loosely" coupled, so that objects can be interchanged or updated more easily.

The benefits of loose coupling are potentially huge and include [Clark96]:

- Higher component reuse
- Higher productivity
- More robust systems, since failures cascade less
- Fewer bugs, as increased reuse means what is reused needs less testing.
- Complex systems become easier to alter, due to higher component reuse.
- Easier component enhancement, modification and bug fixing

Coupling is usually associated with cohesion [Yourdon79], which is a measure of the inter-relationships between functions of a single component. Since our study is to examine distributed systems, we feel cohesion is of secondary interest in this case. Therefore, we concentrate on how component coupling is affected by the choice of mobile code abstraction, and define coupling as:

> "A measure of the external dependencies of a component defined by the number of links that component has to other components within a software system."

4.3 Research Statement

The main aims of the research undertaken in Part II can be summarized as follows:

1] To demonstrate mobile code can be used to build real world software systems

We describe the construction of two prototype mobile code systems. They are used to investigate the effectiveness of the two selected abstractions in building real world distributed systems. To simulate real world software problems the prototypes are constructed to support the Sales Order Process of a UK manufacturing enterprise. This real world business process was identified during an industrial case study (for further details see Chapter 5).

2] To learn how mobile code responds to real software problems

Merely building proof-of-concept systems is a worthy exercise, but systems in the real world very rarely fulfil all the requirements of a business for any length of time. In the majority of cases, the capabilities of a software system will need to be later upgraded to support new functions or features, usually due to a change in a business process. In addition to their creation, we aim to evaluate the prototypes with respect to the issues of understanding and changing a system that currently confront system designers. To achieve this we have extracted several "Scenarios for Change" from data collected during our case study, which will be used to evaluate how well the prototypes respond to change. Three common and related problems facing the software industry today have been identified as candidates for examination. These are:

- **System agility** – how well a system responds to change
- **Semantic alignment** – how well a system embodies the business process it is intended to support
- **Component coupling** - how intermeshed the components of a software system are

From the experiments, we hope to gain an insight into how successful mobile code systems are when subjected to the kinds of pressures prevalent in industry.

However, before proceeding with the construction of the prototype systems, it is important to first examine the technical issues associated with using mobile code. To support our philosophical understanding, we must also appreciate the requirements and consider the limitations of mobile code infrastructures before employing them. For the remainder of this chapter we focus on issues relating to *mobility in the real world*.

4.4 Technical Issues and Enabling Technology

We have seen in Chapter 3 that distributed systems built with mobile code technology usually consist of execution environments that are hosted at different nodes of a network. Mobile agents are able to migrate between these hosts in order to interact locally with static resources and other static agents resident at the hosts. This hosting and migration can be achieved through several different mechanisms, and combinations thereof. In this section, we examine several of the key issues and decisions that must be taken when implementing and using a mobile agent framework.

4.4.1 Strong vs Weak Mobility

The terms strong and weak mobility refer to the method and nature of the mobile agent migration. In strong mobility, the entire computational entity, i.e. its code, data, execution state and program counter migrate to the new host. There are two ways of achieving this, firstly by true migration and secondly by remote cloning. With true migration, the mobile agent is suspended before being transferred in its entirety to the new host. Upon arrival, the agent is restarted and is able to continue its execution at exactly the point at which it was suspended. Remote cloning on the other hand achieves migration by stopping the entity at the first host before creating a copy at the new host. Indeed, some might argue that since computers can only copy and delete [Cox98], both methods are actually the same. Some important examples of mobile agent frameworks that exhibit strong mobility include Agent Tcl [Gray97], Ara [Peine97] and Telescript [White].

Weak mobility on the other hand is only able to migrate the code associated with the entity across the network. Any state or non-constant data that is required by the entity must be packaged up for travel before migration. The onus of this packaging is

placed upon the programmer at design time. Weak mobility is generally easier to achieve technically, especially with programming languages such as Java available, but is burdened by its limitations when complex applications are considered. The programmer must be fully aware of any data that may be required after migration and take care to package it, or it will be lost. The majority of (if not all) mobile agent frameworks based on Java are weakly mobile (see Section 4.6. for examples)

4.4.2 Interpretation vs Compilation

By their very nature, mobile agents are inherently distributed [Clements97]. As such, they must be executable across a variety of platforms and operating systems to achieve their full potential, although in a closed and privately controlled network they may benefit from homogeneity. Their true advantage however, comes from being able to migrate and continue functioning in a heterogeneous network of systems. This advantage is implementation dependent and has greatly influenced the way in which mobile agent systems are created. To enable heterogeneous execution it is usual for these frameworks to be written in some type of script or bytecode that can subsequently be interpreted, usually by a dedicated executing environment. Indeed, the spiralling popularity of Java, combined with its platform independence, has made it the de facto language for mobile agent systems. Interpretation removes the need to recompile an agent at a new host and instead places the onus on merely ensuring an environment exists at the new host that is capable of uniformly executing the agent on arrival. Most examples of this type of system have a server or some type of executing environment in which the mobile agents are executed [Lange98][Gray97]. Interpretation does of course have the previously discussed limitation of execution speed, but this is often seen as a minor trade-off, due to the ease in which portability is achieved.

Compilation is not particularly popular in the field of mobile agents, since it forces the sending machine to be aware of the platform and hardware architecture of the receiver, so that it may choose the appropriate compiler or the appropriate library of native code. As the number of different platforms being supported increases the complexity is wont to spiral out of control. Compilation does however have the advantage of speed of execution. Some examples are [Knabe96] [UCI96].

4.4.3 Resource Management

When a mobile agent migrates to a remote host, any references it has to local resources are likely to become invalid. Before execution can be resumed, all its references must be evaluated and reassigned. This problem can be overcome in a number of ways:

- **Copy** - If the resource can be copied, then the mobile agent can take a copy of the resource with it to the new host.
- **Move** - The mobile agent can take the only copy of the resource along with it.
- **Network reference** - If the resource is static, then the reference can be changed into a network reference that points back over the network to the resource.
- **Reference removal** – If the reference is no longer needed, or cannot be accessed remotely via a network reference, it can be removed.
- **Rebinding of reference** – If another copy or instance of the resource, or a similar resource, is found at the new host, the reference can be rebound to it.

Which tactic to adopt is often determined by the nature of the resource in question, and the programming language being employed. For example, it would be nonsensical to copy or move an entire database to a new host.

4.4.4 Security

Security is one of the most emotive issues raised when discussing mobile agent systems. It is often quoted [Johansen99] as the major reason mobile agent systems have not taken off in the mainstream. There is currently a wealth of research being done on this particular subject [Vigna98]. A brief summary of the most important security issues are describe below in Table 4.

The work described in this thesis is concerned with private networks, in which all the hosts and agents are trusted and their origins known. Thus, the only class of applicable attack is that of a third party eavesdropping on a transmission. This could be overcome by the usual cryptographic techniques employed in such exchanges as email, for example. Therefore, the issues of security are considered external to the scope of this thesis.

Attacked	Type of Attack	Explanation
Host	Host compromised by arriving agent	An incoming agent may try to access and corrupt the host's local files, resources or even try stopping the server in a denial of service attack.
	Host compromised by external third party	Someone who wishes to bring down the host may send a huge number of agents to the host to tie up all the resources, or even crash the host
Agent	Agent is compromised by the new host	If the host is untrusted it may try to access private information, e.g. a credit card number, a password, etc, for later use, or replay.
	Agent is compromised by another agent	During an inter agent conversation the other agent again tries to access private information, or to crash the agent to stop it fulfilling its task
	Agent is compromised by a third party	Since some inter agent comm'n takes places over the network a third party may try to alter exchanged messages for their own benefit, e.g. to recommend their host instead of another, or to reveal content of agent
Network	Network compromised by incoming agent	An incoming agent attempts to flood the network with copies of itself

Table 4. Summary of mobile agent security issues

4.4.5 Communication

Communication among mobile agents in a network can take several different forms. Since there is no guarantee that there is actually another agent at the present node, the most basic inter-agent communication usually begins by using the executing environment to pass messages to another agent. This can be achieved directly, if the agent's identity is known, or can be broadcast to the entire node. Once the presence of the agent is established, communication can then proceed more privately with both agents being involved in a one-to-one dialogue.

Mobile agents are also able to communicate over the network, in a similar way to traditional Internet applications, such as ftp, telnet, etc. Once again, the initial establishment of a dialogue between agents is achieved via the hosting executing environments. Communication with remote mobile agents does have associated problems, caused by the mobility of the agent. Passing messages between two agents requires some type of address, which refers to the receiving agent's location.

Obviously, this can cause problems if the receiver is able to move to a new location, as the address is no longer valid. New techniques for overcoming this particular problem are in the early phases of research and development, but include multicast messaging, where a message is broadcast to the entire network, instead of just to the local node.

At the higher levels of abstraction, communicating mobile agents will usually do so by purely message passing. However, at lower levels of abstraction, for example communicating mobile objects, some sort of remote procedure call mechanism is usually provided, that allows objects to interact in the same manner as contemporary systems.

4.5 Advantages Claimed for Mobile Code Systems

In the previous section, we examined several key technical issues that shape how we may utilise and implement mobile code infrastructures. Simply understanding the technological issues however, will not allow us to make an informed judgement of this new technology. We must also understand what advantages mobility might bestow upon distributed systems built with this new paradigm.

So far, there have been many advantages claimed for mobile agents [Chess97][Lange99]. These claims are usually in the form of qualitative assessments but unfortunately, very few quantitative measures exist to support these claims. However, a summary of some of the more frequently quoted and accepted claims are described in the following sections.

4.5.1 Bandwidth Savings

Distributed systems by their nature are required to communicate over the network. This communication can sometimes be in the form of multiple consecutive interactions between two components, for example, a query client and a database. This type of data querying can result in heavy network traffic. Mobile agents are able to overcome this problem by relocating to the host of the database. Instead of shipping data back and forth across the network, they are able to migrate the required business logic to the data source. Once in situ, they can perform any required queries and process the returned information without saturating the network. After

processing, they are able to continue with their work, transporting merely the result to a new host, if it is in fact needed.

4.5.2 Reducing Latency

Many manufacturing and robotic systems must be controlled in real time. Controlling these systems through a factory wide network can be affected by latency and data timeliness. Mobile agents are able to overcome this problem by migrating to be local to the process and control it in real time, thus bypassing the problems of latency.

4.5.3 Disconnected Operation

As the amount of Internet traffic increases, the response from the telecommunications companies in installing new carrier infrastructure is immense [Kotz99]. Nevertheless, this effort may still not be enough to satisfy the expanding base of users. Moreover, many users will not have access to the high-speed bandwidth available to wealthy corporations. Currently, most home users in the UK still connect via a modem and copper telephone lines. Further, the proliferation of mobile devices, such as palm top computers, which employ wireless networks implies that many users and devices will be extremely limited in the bandwidth available to them. This disparity in quality of connection means that performing tasks that require a continuous connection to the network will be probably not be feasible financially, if not technically.

Mobile agents are a solution to this problem. A particular task can be encapsulated within a mobile agent. The agent is then dispatched to a host that is part of the network backbone, and enjoys massive bandwidth access. Once there, the mobile agent is able to carry out its task in the resource rich environment before returning home. A further advantage of this paradigm is that since the mobile agent is now independent of the device, the device can go offline, or even be switched off, before again connecting later for the agent to return with the results.

4.5.4 Increased Stability

One of the major problems with distributed systems is failure, and the identification of the particular type of failure. Traditional distributed systems are built with the philosophy that the network is permanent, and any failure is unexpected. When it does happen it is very difficult to tell whether the network has failed, the machine that

was hosting the component you were communicating with has died or the component itself has frozen.

One of the underlying philosophies behind mobile agents is that the network is not a permanent resource. By building software with mobile agents, distributed systems can be less dependent on the network, since the underlying tenet is local interaction. Discovering the nature of a failure in a local context is a much easier proposition, and so systems built this way can be more stable. Mobility can also be used to achieve replication for fault tolerance, and support robust distributed systems. If a host is being shut down, or experiencing problems, an agent is able to react to this by migrating to a new host where it can continue with its operations.

4.5.5 Server Flexibility

In contemporary distributed systems, when data is exchanged between communicating hosts, each host owns a copy of the code that is required to package outgoing and interpret incoming messages. As protocols are evolved to better support efficiency and security, the effort required to upgrade protocols becomes immense. By using mobile agents, the protocols can be encapsulated within the agents, and removed from the servers. Thus, if a protocol requires an upgrade the mobile agent population can be upgraded gradually as and when required, instead of the entire server base.

Further, since mobile agents are able to carry around their own code, the distributed system can become more flexible since the mobile agent is not merely limited to the functions a server predefines. It is able to bring along new or improved code and can extend the functionality of the server in which it is executing.

4.5.6 Simplicity of Installed Server Base

An additional advantage of relocating the computational logic and protocols within the mobile agent is that the installed servers become much simpler. Effectively, a server becomes merely an executing environment for hosting mobile agents. As this requires far less functionality pre-engineered into the software from the outset, it can help with preventing legacy. Further capabilities can be added by mobile agents at a later date.

4.5.7 Support distributed computation

Mobile agents are inherently distributed, and as such can be a fundamental enabler for distributed computation. However, they are also heterogeneous, often separated from both hardware and software dependencies by their executing environment. This means they are an ideal technology for integrating disparate legacy systems that have dependencies already.

4.5.8 Commentary

The advantages we have seen described for mobility are certainly exciting. Whilst very few quantitative results exist to verify the claimed advantages, the overall picture painted is one of a completely new paradigm for building distributed systems. Such is the excitement that many research labs have already begun to produce mobile code infrastructures [Lange98] [Concordia]. In later years, this initial group may become known as 1st generation infrastructures.

As the mobile code research field has matured, a few quantitative measures are beginning to be published [Picco98b]. Papastavrou *et al* [Papastavrou99] have shown that using mobile agents to perform your database queries locally can have a dramatic affect on system performance. Johansen has shown that bandwidth usage can indeed be reduced by significant levels by using mobile agents when compared to traditional client/server architectures [Johansen99].

It is the author's belief, however, that the majority of advantages discussed in the previous sections are merely optimisations. Many of these advantages could be achieved with contemporary distributed systems, for example by redesigning communications protocols. The true advantage of this new paradigm is the types of distributed system that can be built: ones that do not suffer from the Waldo problems. In the next section, we review some of the well-known frameworks to see how these new abstractions are manifesting themselves.

4.6 Survey of Mobile Agent Systems

The rapid explosion of interest in this field of research means that there are a large number of new mobile agent frameworks appearing, almost continually. The Mobile Agent list [MAL99] currently numbers the known packages at 64. In this section, we

review some of the better-known frameworks and analyse how they embody the mobile code abstractions discussed in Chapter 3.

4.6.1 Java

Although not marketed as a mobile agent framework, the Java [Gosling96] Development Kit does provide enough native facilities to support weakly mobile code. This should not be a surprise since the original goal of Java's designers was to provide a portable, easy to learn, network aware object-oriented language. To ensure portability, Java was designed to be platform independent. Instead of compiling Java into native instruction codes, it is compiled into an intermediary format known as bytecodes. The bytecodes can then be interpreted on *any* platform that has a suitable java interpreter; the interpreter is known as the Java Virtual Machine (JVM) [Lindholm99]. By having the intermediary bytecode stage, Java is an ideal language for weak code mobility. The most widely known examples of Java's mobile code capabilities are probably applets and servlets [Hopson96], mobile snippets of code that can be transferred over the network in an asynchronous manner. Applets and servlets should not be viewed as mobile agents however, since they are merely single-hop pieces of code that contain no notion of autonomy. They do embody the Remote Computation (RC) and Code on Demand (CoD) design abstractions (see Section 3.4).

Inherent platform independence supported through interpretation has made Java an extremely popular choice among mobile agent framework implementers. One might even argue it is the de facto language. These facilities in conjunction with its security model [Gong99] and object serialisation [Sun98b] make it a particularly useful technology base from which to begin.

4.6.2 D'Agents

Developed at Dartmouth College, D'Agents [Rus97] is one of the new breeds of mobile agent framework. In its first incarnation as Agent Tcl [Gray97], D'Agents employed a Tcl [Ousterhout94] interpreter, extended to support strong mobility. When an agent wishes to migrate to another machine it need only call a single function, `agent_jump`, which triggers the interpreter to package up the complete state of the agent and send it to a destination machine. Strong mobility has always been a design goal of the Dartmouth Group and recently, D'Agents has been updated to be a

multi-language framework and now supports strong mobility in Java. However, this facility has come with a price; in order to support strong mobility in Java the D'Agents team had to modify the JVM, which means that the framework will only work with the specialised JVM. With the current rate of change in the Java world, this means that the D'Agent interpreter can quickly become out of date.

4.6.3 Mole

Mole [Straßer96] was the first mobile agent framework developed in Java, and was initially released in 1995 by the IPVR group of Stuttgart University. Mole supports weak mobility only, a choice the designers justify in [Baumann97]. Interestingly, the Mole group assert that their choice of weak mobility was to avoid the problems of using a modified JVM that quickly became out of date. Their goal was to provide a pervasive framework the worked 'out-of-the-box' with any standard JVM. This is in contrast to the D'Agents group and demonstrates the generally unexplored nature of the research field. Whether strong or weak mobility is the correct methodology remains an open question within the mobility community.

Mole provides the notions of *places*, the executing environment, where *user agents* are able to meet and communicate. They can interact with the underlying operating system resources via *service agents*, which are always stationary. Mole supports a number of communication mechanisms including *badges*, *sessions* and *events*. An ascending hierarchy of increasingly anonymous and wider scope of influence mechanisms, they are fully described in [Baumann97].

4.6.4 Hive

Hive is a distributed agents platform, a decentralized system for building applications by networking local system resources, and taking advantage of mobile code [Minar99]. Its designers, a group at the MIT Media lab, are using it to provide the infrastructure for connecting their many Things That Think [Gershenfeld99] research initiatives. Hive is built using the standard Java features of object serialisation and interpretation used by so many mobile agent frameworks and therefore supports weak mobility.

The Hive architecture consists of the following three abstractions: *cells*, *shadows* and *agents*. A cell is the executing environment in which agents are hosted. Cells also contain shadows, which are placeholders for local resources, for example a display or printer. The designers of Hive have made particular efforts to address the problems of agent description and Hive supports both a syntactic and semantic ontology.

Inter-agent communication in Hive has been achieved by using RMI as the communication mechanism. This allows the methods of Hive agents to be executed remotely. While this approach is simple, and uses built in capabilities of the Java language, it has the disadvantages of loss of control and security. In the author's opinion, it also blurs and lowers the abstraction level of the mobile agent to one of merely a mobile object. If an agent's methods can be called and executed remotely, then any notion of autonomy for the agent has been lost. Hive thus embodies a hybrid abstraction, drawing elements from the autonomous agents research arena, and from contemporary RPC distributed systems. This hybrid abstraction has caused the Hive team some considerable headaches in achieving their goals [Minar99b]. This is a shame, since the ontological descriptions supported by Hive are superior to many if not all of the other frameworks reviewed.

4.6.5 Voyager

ObjectSpace's Voyager platform is a one-size-fits all communication infrastructure. At the time of writing Voyager currently supports EJB [Sun99], CORBA, DCOM, and RMI. In its early days ObjectSpace promoted the capability of Voyager to take existing CORBA IDL classes and "virtualise" them, effectively making them weakly mobile. This was a major selling point for Voyager, but recently the company has been playing down these capabilities [Glass99]. Voyager should really be viewed as a Java based messaging broker that has some added capabilities from the mobile agent field. This allows programmers to create network applications by choosing between traditional and mobile distribution technologies, and has been a widely successful product.

4.6.6 Jini

Jini [Arnold99] is Sun Microsystem's proposed architecture for embedded network applications. It is built using Java and RMI in much the same way as Hive. Jini

provides simple mechanisms that enable devices to plug together to form an impromptu distributed system. Each device provides services that other devices in the system may use. These devices provide their own interfaces, which Sun claims "ensures reliability and compatibility". Much to the chagrin of the Hive team, Jini is a very similar framework, although it does not have the shadow/agent conceptual split. Most important however is that Jini's creators do not consider location to be an important part of the abstraction. Where a particular service resides in the network is not of importance to Jini, the interfaces and lookup services are intended to handle this sort of issue. Further, Jini only supports single-hop mobility, and as such can be categorized as embodying merely the CoD abstraction. This continued support of the location transparency abstraction and only a basic mobile code abstraction are surprising as Waldo is one of the authors of the Jini specification.

4.6.7 Aglets

The Aglet Software Development Kit (ASDK) [Lange98] has been developed by IBM's Tokyo Research Labs, and was one of the first and most publicised Java based mobile agent frameworks released. The core abstractions supported by the ASDK are that of an *aglet*, a *proxy* and a *context*.

An aglet is a mobile autonomous agent, whose structure can be considered to consist of two distinct parts, the aglet core and the aglet proxy. The core is the heart of the aglet and contains all of the aglet's internal data and logic. It provides interfaces through which the aglet may communicate with its environment. The aglet core is then encapsulated by an aglet proxy that acts as a shield against any attempt to directly access any of the aglet's private internals, and can hide the real location of the aglet from malicious aglets.

The aglet context is the executing environment in which the aglets exist. It provides an interface to the underlying operating system through which aglets are able to access core facilities, and gain references to other aglets' proxies. The context also manages the lifecycle of an aglet. Since the ASDK only provides weak mobility, this lifecycle is one of the ASDK's most valuable features since it allows the programmer to describe behaviour an aglet should perform in reaction to certain events, for example, the shutdown of the current host, or a request to migrate to a new host. This

lifecycle is supported through an event-based scheme that is well known in the window system programming world. Aglets implement a number of event handling methods that can be customized by the programmer. These methods cover all the important events in the life cycle of an aglet (creation, dispatch, arrival, deletion, etc.). For example, if you move an aglet it will be notified upon leaving its host and upon arrival at the new host. Of all the frameworks reviewed, Aglets enforces the mobile agent abstraction and metaphor most strongly. In contrast to Hive, all communication between aglets is via messaging. On receipt of a message, an aglet is able to decide what to do with the message, and when, thus sustaining the autonomy of the agent.

4.6.8 The Mobile Agent Graveyard: Telescript and Odyssey

Developed by General Magic Telescript [White96] was an object-oriented programming language designed for the development of Personal Intelligent Communicators (PICs). PICs were defined as being handheld palmtop-like devices with little memory and low bandwidth capability. Telescript was the first of its kind to appear and ground breaking in the facilities it offered.

Telescript was an interpreted language that supported strong mobility. There were actually two levels of the language: *High Telescript,* the actual language used for implementation, and *Low Telescript,* a Postscript like language which could be interpreted better by the top level executing environment, the *engine.*

Other abstractions supported by Telescript included *agents,* mobile agents that were able to migrate on a single command of `go`; *places,* stationary processes that provide interfaces to services, and were normally inhabited by agents; *tickets,* objects that describe an agents journey; *permits,* objects that define the capabilities and resource constraints of an agent.

There is an important programming paradigm difference between Aglets and Telescript that demonstrates the differences between strong and weak mobility: Telescript is focused on process migration that allows you to "go" in the middle of a loop and resume the execution in the middle of that loop on another machine. Aglet developers must consider how to deal with migration of non-static data.

Sadly, Telescript is no longer available, having gone to the Mobile Agent Graveyard[7]. Odyssey was General Magic's attempt to revive its flagging fortunes with a Java based mobile agent framework that resembled Telescript. It never made it out of beta.

4.7 Choosing a Mobile Agent Framework

Whilst there are an increasing number of mobile agent frameworks, when the study described in this thesis began the choice was limited to perhaps half a dozen. From those available, IBM's Aglet framework was selected. It would be appealing to be able to demonstrate a methodology employed for selecting the framework, but there is none. The Aglets package was chosen due to the connections of Danny Lange, the inventor and chief architect of Aglets, to researchers at MSI. However, in defence, several important factors support the choice of the ASDK:

- it was one of the first to use the Java programming language;
- it contains the notion of agent itinerary which systems such as Telescript did not support;
- it is being proposed for submission to the Object Management Group (OMG) Mobile Agent Facility RFP;
- it includes a fine grained security model
- aglets has proven to be an extremely popular framework in the mobile agent community for its clear agent abstractions and lifecycle facilities

Actual mobility in the ASDK is enabled by the provision of two facilities:

- the Agent Transfer Protocol (ATP)
- the Java Agent Transfer and Communication Interface (J-ATCI).

The ATP is an application level protocol for distributed agent based information systems and facilitates migration of the aglets over a network. Based on the naming conventions of the Internet, ATP uses the Universal Resource Locator (URL) [Berners-Lee92b] for specifying host locations, whilst maintaining a platform independent protocol for enabling the transfer of mobile agents between networked computers. Although this protocol has been released with the ASDK, its domain of use is by no means exclusive to aglets, as it offers the opportunity to handle mobile

[7] It lives on though, through furtively copied gold CD's!

agents from any programming language and a variety of agent systems, as long as they implement the protocol interfaces.

Reinforcing the ATP at a higher communication level is J-ATCI, an independent agent protocol enabling agents to move and communicate within a network. J-ATCI is a simple and flexible programming interface that enables programmers to develop platform independent agents without having to build into them the necessary protocols for wire communication. By ensuring a native implementation of the J-ATCI designers can expect their agents to function on any platform. The J-ATCI has also been submitted to the OMG.

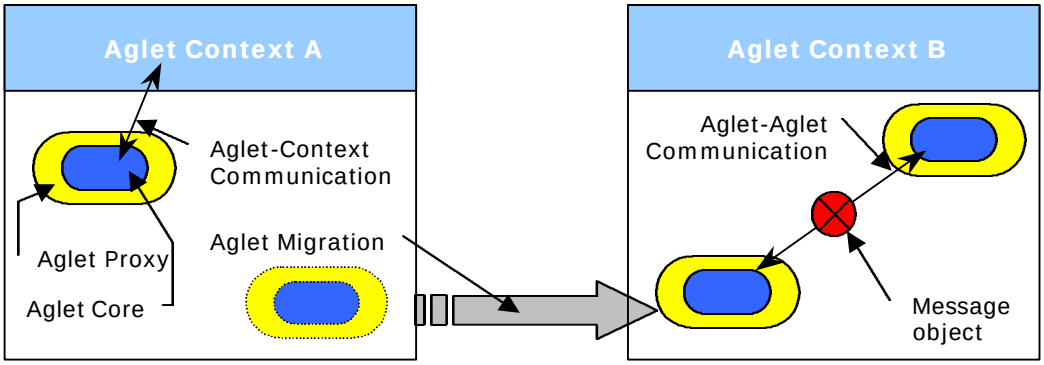


Figure 18. The Aglet Environment

4.8 Concluding Remarks

Pure academic thought might have been encouraged in the classical world, but in ours, we require facts too. To support the philosophical argument of Part I, we construct two prototype distributed systems with mobile code technology. To evaluate the systems we have identified several issues that are constantly engaging the software industry: system agility, semantic alignment and component coupling. The business process our systems are intended to support has been extracted from an industrial case study. The prototypes will be subjected to several *Scenarios for Change*, which will allow us to gain an insight into how well they perform.

This chapter also contains a review of the technical issues involved with implementing the mobile code abstractions, a summary of many of the claimed advantages for mobile code and a roundup of several of the more established mobile code infrastructures. In the following chapters, we report on the implementation and

evaluation of our prototypes. Before that however, we describe the case study that was used to generate a business model and process for the prototypes to support.

5 I.T.L. : An Industrial Case Study

5.1 Introduction

This chapter describes the industrial case study undertaken in the course of the research described in this PhD. It was performed at Instrument Technology Ltd (ITL), a high performance vacuum component manufacturer based on the south coast of the UK, in Q1 1997. In the next section, we discuss the methodology and the objectives of the case study.

5.2 Why a case study?

> "A case study is an exploration of a question or phenomenon when little is known in advance, and where the situation may be complex." [Yin94]

Case studies are able to examine processes within a specific context, draw on multiple sources of information, and relate a story, usually in a chronological order. In case studies, we are able to ask: "How or why does this occur?" We can create a rich, textured description of a social, economical or infrastructural process [Scanlon97]. This information can give an insight into how to gain answers to more specific questions, or produce conceptual models of a business process.

It has already been shown that the mobile code community recognises the lack of real world examples of their technology [Picco98] [Milojicic99]. We aim to prove that mobile code can be used to build real software systems. Therefore, the scope of this particular study was to gain an insight into I.T.L. and identify a suitable business process. The extraction of an industrial process model would provide a suitable reference around which the subsequent prototype implementations could be built. Further, the case study allows us to generate real world scenarios that can be used to evaluate the prototype systems after their construction.

When performing a case study it is extremely important to select an appropriate methodology [Jones97]. To achieve our objectives, the methodology selected was to carry out a qualitative, exploratory case study. Qualitative studies are particularly useful in attempting to answer questions such as 'Why?' or 'How?' [Strauss90], while exploratory studies are those that attempt to gain an initial insight into a situation.

Together they allow the examiner to create a 'snap-shot' in time of a particular process or situation. The methodology was considered appropriate, as it was capable of fulfilling our requirements:

1] To produce an SOP model,
2] which was based on a real world example,
3] upon which a set of experimental scenarios could be based.

The models generated from the case study are presented and discussed later in the chapter, following an overview of I.T.L.

5.3 Who are I.T.L.?

Instrument Technology Limited (I.T.L.) is a British manufacturing company based in East Sussex. It has been established for over twelve years, and usually performs steadily. A recent diversification in product range had reaped benefits however, and at the time of the case study, the company had shown a growth in turn-over from £500k to nearly £10m in five years, whilst concurrently developing an extensive, global customer and distributor base. More recently, the company has been affected by the crash of the Asian tiger economies.

5.3.1 What does I.T.L. do?

I.T.L.'s core business is manufacturing high performance vacuum components, primarily for the semi-conductor industry. The scope of the product range ensures that there are few other companies in the world that manufacture a greater diversity of standardised vacuum components. At the time of the case study, there were over 2,000 modular products and almost 7,000 items in the product catalogue. In an interview with the managing director [Barlow97] it became clear that these figures were expected to increase. The company has been quick to recognise the trend towards customer-driven specialised services and part production. This is supported by an extremely flexible design service offering almost unlimited choice to customers, who are able to submit their own specifications for product manufacture. Co-existing with the standardised product group is the specialised vacuum chamber division, which builds intricate, high pressure chambers and vacuum chambers, usually for advanced research facilities such as CERN.

5.3.2 How does I.T.L. work?

Until 1997, I.T.L. perceived[8] its largest market to be in the UK and Export direct sales, in which they have a substantial market share. However, the emphasis for the company is now shifting to much larger, more lucrative contracts with several international OEM's. Deals with a number of multinationals have consolidated previously successful working relationships, and ensured good market standing for I.T.L., which is now emerging as a global “player” in the vacuum component market. For direct sales, a network of Sales agents deals with the promotion and marketing of brand products. The network encompasses Europe, the Far East and Central and Southern Africa, with several more slated for adoption in the short term. All orders are still supplied from I.T.L.’s headquarters in the UK. OEM partners are offered exceptional configurability in delivery and service. For example, specialised packaging, branding or invoicing.

Figure 19. An overview of I.T.L. around the world.

I.T.L. now perceives the greatest potential for sustained growth in expanding its network of Sales Agents into new markets, whilst attempting to broker new OEM deals with further American companies [Barlow97]. Consolidation with its oriental partners has also brought new opportunities in reducing manufacturing costs, and the company is investigating the viability of investing in new manufacturing facilities in the Far East.

[8] The term “perceived” used here is factually correct, at the time of writing no one at I.T.L. was able to give exact figures for any of their markets.

Finally, I.T.L. has settled on a long-term strategy of expanding its global presence. In doing so, I.T.L. has realised that it will no longer be economical to continue with centralised stock control since transportation of its products is expensive. Ergo, the company is considering adding new stock control centres or warehouses at globally strategic locations.

5.3.3 Commentary

With the increasingly extensive portfolio of products and parts, the configurability that I.T.L. offers to its customers, coupled with the long term strategy of expansion and the need to remain responsive in the market place, it is clear that I.T.L. requires a high degree of flexibility from both its business practices and the supporting IT infrastructure.

I.T.L. is also hoping to expand both its network of Sales Agents and its stock control centres. This requires a radical change in the company's business practices. It must transform from a central and localised operating model to a distributed one. The pitfalls and problems associated with transformations of this kind are well documented [Peters82] [Hammer93] [Goldman95].

Equally, as the Asian Tiger economies example demonstrates, I.T.L. is competing in a fluctuating market. Responding to such problems as, for example, changing suppliers or meeting 'Just In Time' (JIT) manufacturing requirements mean the company must strive to remain agile. Here, agility is considered the ability to respond quickly to market pressures. For example, both up and downturns in orders, adding or removing suppliers, adding or removing sales agents, etc.

It was our aim to generate a process model from a real company. This would then form the basis for our implementations, and would allow us to evaluate their performance when subjected to the kinds of pressures a real software system may experience. From the case study, it is clear that I.T.L. is a prime example of a manufacturing enterprise facing the very real pressures of remaining agile and competitive. The requirements of I.T.L. can be summarised as:

- It requires a high degree of flexibility in its IT infrastructure
- It must be able to add new sales agents quickly

- It needs to add new stock control centres
- It must be able to upsize and downsize with equal ease

5.4 Process Modelling

Having established I.T.L. was a suitable candidate upon which to base our implementations it was important to identify a suitable business process. The requirements of I.T.L. listed in the previous section all pertain to the Sales Order Process (SOP). Indeed, the SOP plays a pivotal role in any business that relies on constant orders for survival, and involves links to customers, distributors and suppliers throughout the world. This is a perfect process to support with a distributed software system, and therefore, the decision was taken to use I.T.L.'s SOP as the process model.

Understanding the internal process of a company can be complex. A simple but effective tool that is often used for this purpose is a data flow diagram (DFD) [DeMarco78]. Using DFDs, the core business processes of I.T.L. were modelled in an attempt to understand how I.T.L. responds to a new order (see Figure 20). In this diagram, the many processes are defined by the senior management figures that are responsible for those particular areas. Each core process is surrounded by a dotted line for further clarification.

From this rather complex diagram, it is possible to extract the core business processes and represent them in a higher level, abstract view. Figure 21, the Abstract Process Model (APM), shows this simplified view and depicts the interactions between the each process upon receipt of a new order. The decision branch shown in Figure 21 has been intentionally omitted from Figure 20 for reasons of clarity. By examining the interactions between the major components of the APM a basic visual model was generated to represent the entire process. This can be seen in Figure 22. To better understand this model we will walk through an example of a new order being placed.

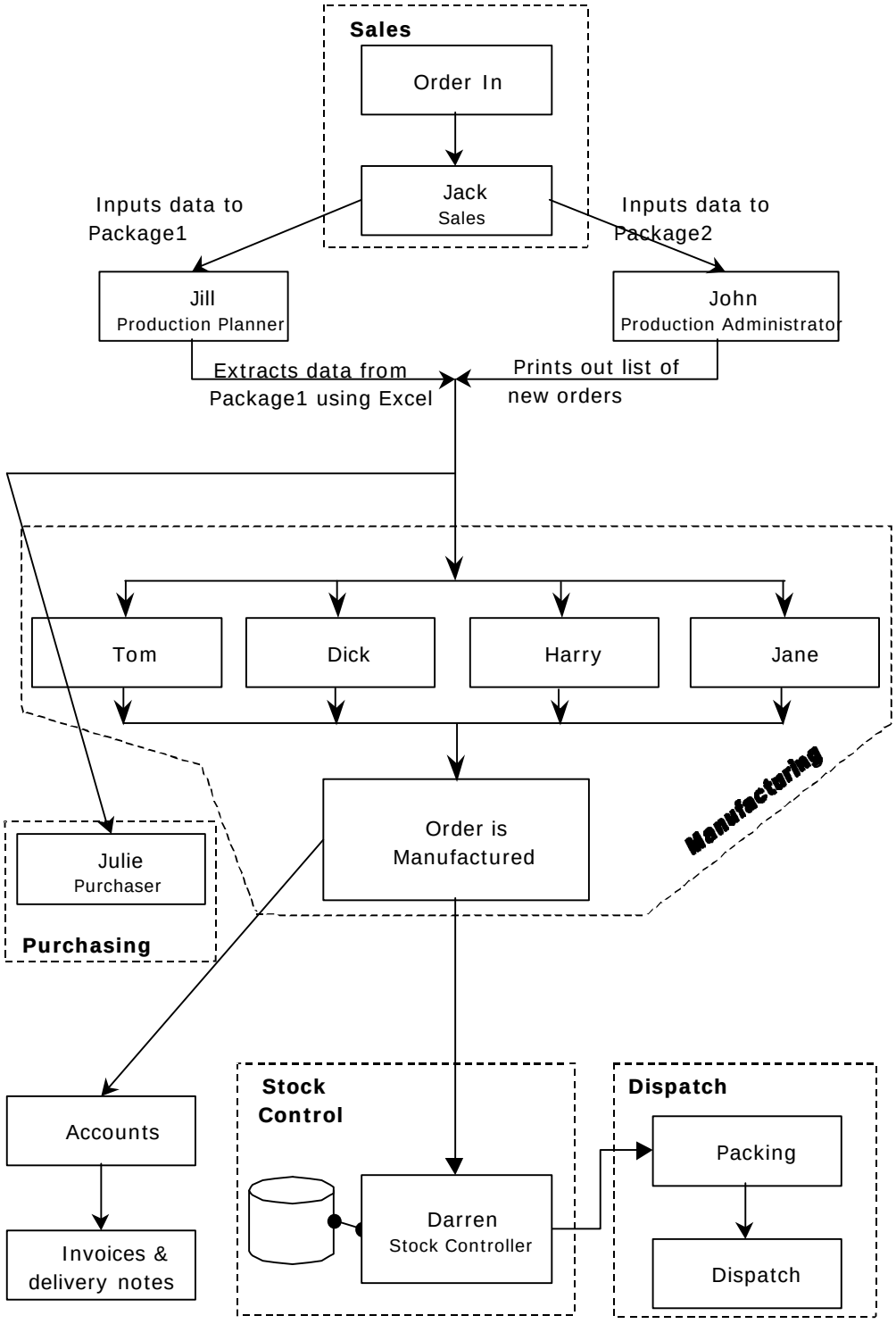


Figure 20. Information flow through I.T.L. on receiving an order

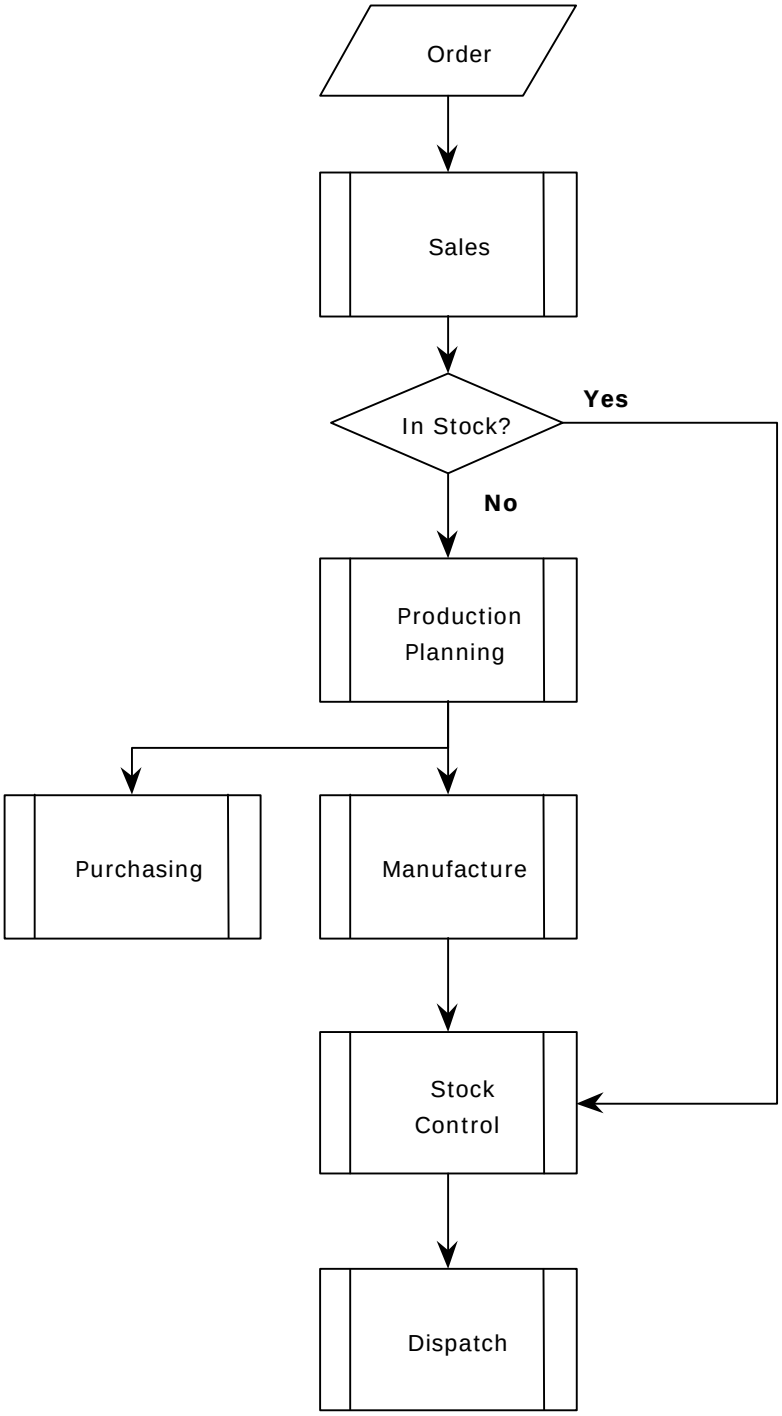


Figure 21. Abstract Process Model

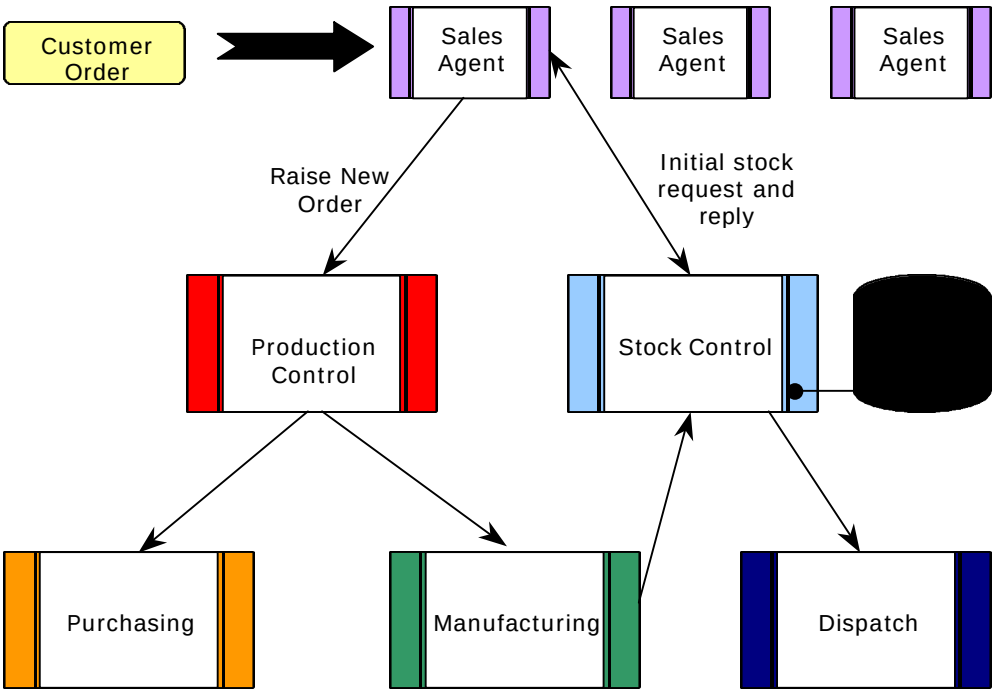


Figure 22. The Sales Order Process

5.4.1 A Walkthrough

A new Customer Order is placed with a Sales Agent. The Sales Agent then interrogates Stock Control to see if the order can be fulfilled from the existing stock. If it can, a new Order is raised and the items are allocated to that order number before being dispatched to the customer, along with an invoice.

If the items are not in stock, then the order is passed to production control where again, an Order is raised. Accompanying this Order is a new Works Order for the required manufacturing of the requested products, or product parts. The Works Order is then passed to manufacturing for completion, and if necessary purchasing for replacement of raw materials. Once the product or parts are completed, they are booked into Stock Control before being checked out again for dispatch. The standard delivery time at I.T.L. is three weeks, unless the order is being specially manufactured to specifications submitted by the customer.

5.4.2 Refining the Model

Implementation of two software systems to support in full the Sales Order Process of a manufacturing enterprise is beyond the scope and time frame of a PhD. Therefore,

we decided to concentrate on the interactions of sales agents handling order requests and the stock control centres. These particular facets are fundamental to the SOP as a whole, and are intrinsically associated with the issues of building distributed software systems. Thus, these processes form the major components of the subsequent prototype implementations.

The Production Control process was removed from the model since scheduling is an entire field of research in its own right and was deemed external to the objectives of this thesis. In addition, the greyed out areas of Dispatch and Manufacturing represent processes that were considered of secondary importance to the requirements identified in Chapter 4. These would make excellent candidates for investigation and expansion in any future work. The finalised model used in the implementation can be seen in Figure 23.

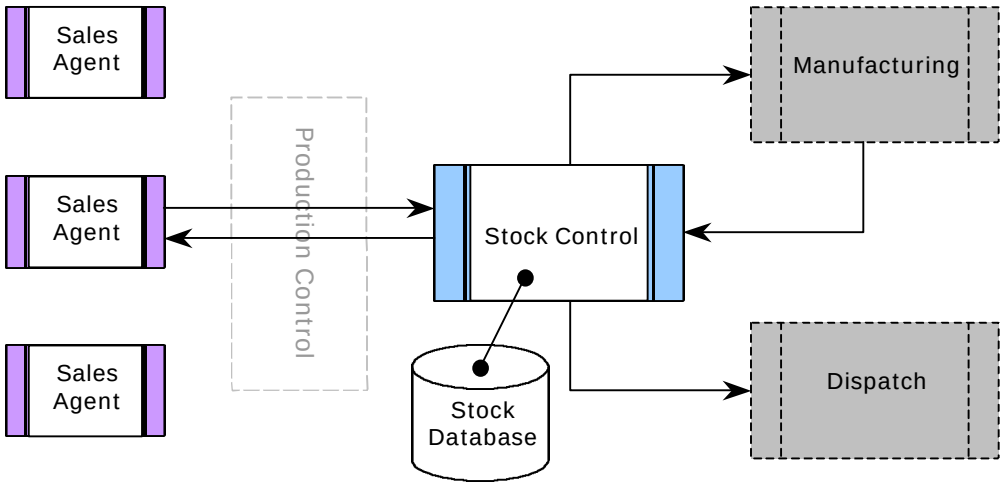


Figure 23. Modified Sales Order Process model

5.5 Concluding Remarks

Gaining an insight from a real world manufacturing enterprise is an invaluable tool for developing a model from which to base experimental work. This chapter has presented the case study undertaken at the vacuum component manufacturer Instrument Technology Ltd. Examination of I.T.L.'s core business processes has yielded a high level abstract model based around the Sales Order Process. This model will be used as the basis for the prototype implementations described in the next chapter. In addition, the company's background and operations were examined,

resulting in the identification of a set of requirements that I.T.L. had of their software system. These are summarised below.

I.T.L.:

- requires a high degree of flexibility in its software systems
- must be able to add new sales agents quickly
- needs to add new stock control centres
- must be able to remove new additions with equal ease.

In the next chapter we describe the implementation of our two prototype systems.

6 Implementation

6.1 Introduction

It has been stated that the field of mobile code research lacks examples of real world applications [Picco98]. Therefore, the work in Part II of this thesis has been undertaken with that fact in mind. In support of our philosophical argument for mobile code, we wish to demonstrate the feasibility of actually building real distributed systems with this technology.

In the previous chapter, we described the generation of a Sales Order Process model, which we aim to support with mobile code technology. We have further refined the model to focus our investigative work on those aspects that depend on distribution by choosing to concentrate on the interactions of sales agents dealing with order requests and the stock control centres.

In this chapter, we describe the implementation of our two prototype systems, a mobile object version of the business model and a mobile agent version. First, we begin by presenting a top down view of the implemented SOP model, before going on to discuss the common parts of the two prototype systems and detail their differences.

6.2 The Model

Figure 24 depicts the implemented mobile agent model of the SOP. The fundamental operation of the process is as follows: following an enquiry from a customer to a SalesAgent (SA), an OrderAgent (OA) is dispatched to the StockControlAgent (SCA) where it requests the fulfilment of its order by passing an Order object. The StockControlAgent, which is resident at a distribution point, queries the stock database to see if enough products are in stock. If there are enough products, the StockControlAgent then returns a DeliveryDate object to the OrderAgent. The OrderAgent then returns and reports to its parent SalesAgent, which is then able to notify the customer of the delivery date.

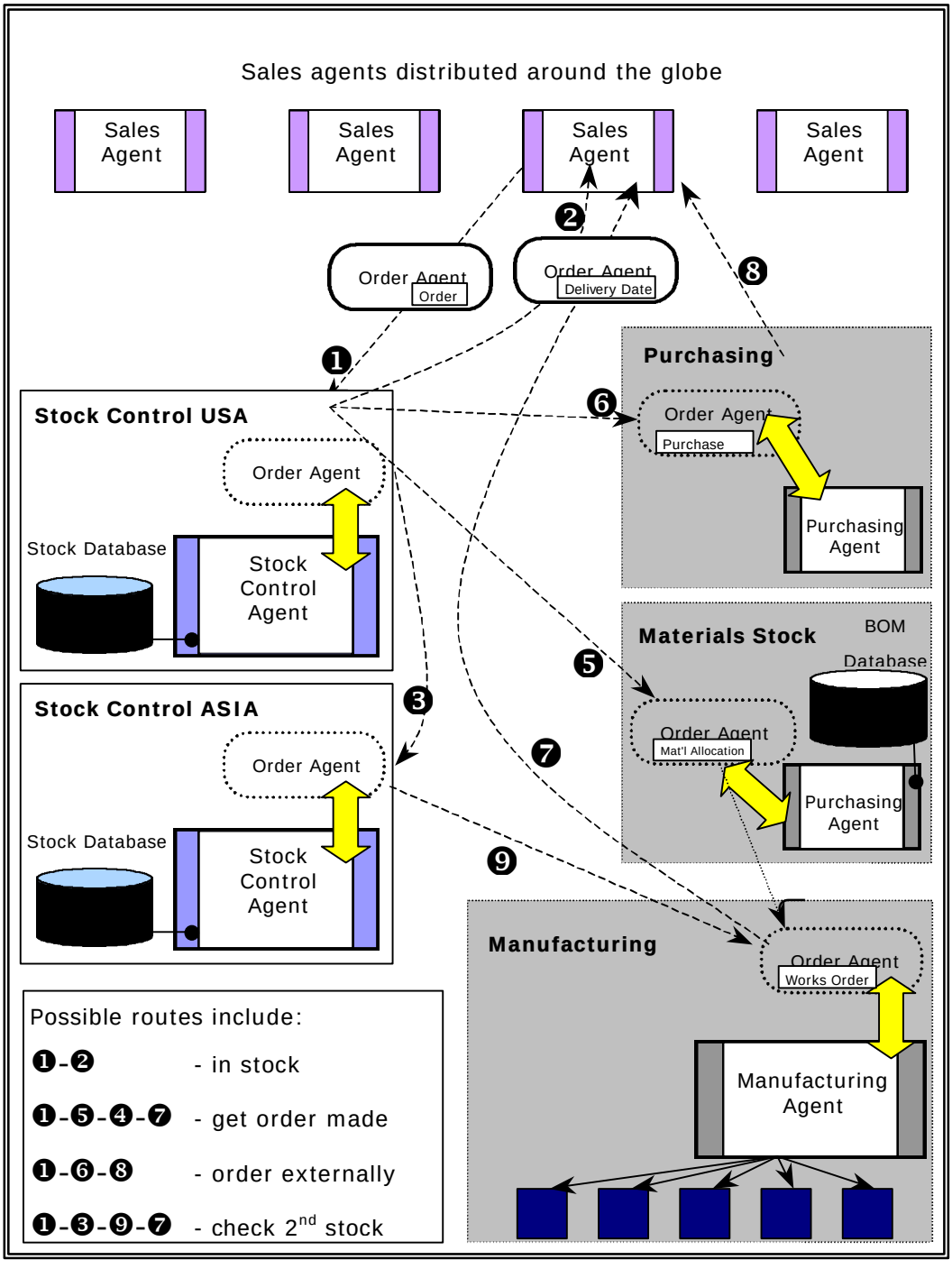


Figure 24. Agent Sales Order Process Model – with example routes for OrderAgents

If there are not enough products in stock to satisfy the order, the OrderAgent migrates to the manufacturing plant where it uses the Product ID encapsulated in the Order object and queries the BOM database for a list of sub-parts or raw materials required. This is then encapsulated within the OrderAgent, which is dispatched to manufacturing to deliver it, before returning to the SalesAgent with a DeliveryDate

object containing a standard delivery date. If there are not enough raw materials in stock, agents within the manufacturing plant server generate a PurchaseOrderAgent that encapsulates details of all the required materials.

The mobile object model is very similar to that described above, the key difference being that the results from stock database queries are gathered from remote StockControlAgents by a mobile OrderObject guided by a specific itinerary. Instead of processing this information locally to the data source, it is returned to the SalesAgent for processing. At arrival, the OrderObject delivers the results before being terminated. If further excursions are necessary, the SalesAgent creates new mobile objects and dispatches them as required. The mobile object does not make autonomous decisions based on the acquired information.

6.3 The Bestiary

The implementation work described in this thesis was undertaken using IBM's Aglet Software Development Kit [Lange98], a mobile agent development framework that was extensively described in Section 4.6.7. This framework has been used as the base upon which to implement the two different versions of the SOP model. Each major process has been embodied as an agent, and there is quite a large overlap in commonality between the two systems. Similar amongst both models are the static agents consisting of SalesAgents, StockControlAgents, ManufacturingAgents, PurchasingAgents and DispatchAgents. As one might expect, there are also mobile components to the systems, and it is here that each system differs from the other. In the mobile agent system, there are OrderAgents, whilst in the mobile object system there are OrderObjects. Generically, we will refer to these as the Order components of the systems. This is primarily, although not entirely, where the distinction between the Remote Computation and Mobile Agent abstraction is evident. It should be noted that in a static analysis of the system, the mobile Order components are a single entity in the design. However, during execution the number of migrating mobile components in the system would be significantly more than the number of static components. In the following sections, we discuss each agent type and its relationship to other agents.

6.3.1 OrderAgents

OrderAgents represent the mobile components in the Mobile Agent system. The agents discussed in this paper can be classified in line with Franklin and Graesser [Franklin96] as goal oriented, communicative, and mobile i.e.:

- **Goal oriented** – they do not simply act in response to the environment
- **Communicative** – they are able to communicate with other agents
- **Mobile** – they are able to transport themselves from one host to another.

On creation, each OrderAgent is given a copy of a new `Order and` an `Itinerary` that contains details of which hosts they must visit to enquire about completion of their Order. Encapsulated within the `Itinerary` are `Tasks`, which the OrderAgent carries out on arrival at a new host. Once the OrderAgents have been given an `Order`, they are then responsible for completion of that order. Some example program listings of an OrderAgent and a Task can be found in the Appendices.

After creation, the OrderAgents migrate to the first host in their `Itinerary` to interact with the resident StockControlAgent. This interaction will involve the OrderAgent querying the StockControlAgent as to whether the Order it is carrying can be satisfied by the levels of stock currently held. The actual stock database is queried by the StockControlAgent; the OrderAgent does not interact with it. The OrderAgent processes the results returned by the StockControlAgent. If the relevant stock is available the OrderAgent asks the StockControlAgent to book out the stock to its Order number before returning to the SalesAgent that created it to report on the delivery date, whilst the StockControlAgent sends a message to the DispatchAgent with details of where to send the products. If the stock levels at the first StockControlAgent are unsatisfactory, the OrderAgent is able to migrate to the next host in its list to begin the process again. However, if no StockControlAgents are able to satisfy the Order then the OrderAgent will proceed to the ManufacturingAgent to request production of the relevant components. Although this behaviour remains unimplemented, it is intended that the ManufacturingAgent would then interact with some scheduling software system to ascertain an estimate on the required time for manufacture that the OrderAgent could use to report to the SalesAgent. Currently, this communication consists of a simple message and acknowledgement from the ManufacturingAgent.

The valid outcome for the goal of the OrderAgent is reporting a delivery date for the order to the SalesAgent. If all else fails, it will return and report that it has failed, allowing the SalesAgent to begin the process again. In the future, this may also include reporting an allocation for raw materials, an internal works order number and time to manufacture. While not complex, OrderAgents usually make up the majority of the agent population in the system, although this is dependent on the number of enquiries received by the SalesAgents. Potentially, there could be hundreds of mobile OrderAgents migrating through the network, attempting to fulfil their own particular `Order`. Since OrderAgents require no interaction with a user, they have no Graphical User Interface (GUI).

6.3.2 Order Objects

OrderObjects are the mobile components of the Remote Computation system. However, in contrast to the mobile agent system, it is more appropriate to view the mobile objects as mobile messengers. Initially they appear to perform the same function as the OrderAgents described above, and in many respects, this is true. On creation, the OrderObjects are given an `Itinerary` and an `Order` and are dispatched to the first host on their list. There, they again query the StockControlAgent to establish whether the order may be fulfilled at that host. Although OrderObjects are still able to migrate to a data source and take advantage of local interaction and all the advantages that brings, they do not contain the business logic to autonomously process any results. They merely add them to their records before migrating to the next host in the `Itinerary`. Once all hosts in the list have been visited, and all stock databases queried, the OrderObjects return to their origin to report the findings to their parent SalesAgent, after which they are terminated. In this system, the processing of the results is performed by the SalesAgent, which creates a new OrderAgent and dispatches it to one of the hosts to commit the stock to the Order. Again, during execution there may be many hundreds of mobile OrderObjects instantiated within the system.

6.3.3 SalesAgents

SalesAgents are static agents that are responsible for generating Order components, giving them an `Order` and `Itinerary`, and sending them out into the network so they

may interact with StockControlAgents. SalesAgents are the human users' main interaction with the SOP system and therefore they have a GUI with which the sales person can create a new Order. SalesAgents are more complex than the Order components, since they must keep track of current orders, but they still remain "slim" and can be manifest as a client for sales persons working on terminals or NetPCs, or be hosted on a laptop for travelling sales persons.

In the mobile agent version the only logic contained within these agents is that required to create a new OrderAgent, with its accompanying `Order` and `Itinerary`. They are capable of maintaining a list of spawned OrderAgents, and thus are aware of which Orders have been fulfilled. In the mobile object version, they also contain the business logic required to process the results returned by their slave OrderObjects.

6.3.4 StockControlAgents

The StockControlAgents are another example of static agents within the systems, but as they do not interact with human users, they have no user interface. They are responsible for handling all requests for products and materials made by the Order components, and act as custodians for the information contained in the stock databases. As such, they are a communications bridge between the data sources and the other agents in the system. All requests for stock levels and allocation must be made through the StockControlAgents.

Manufacturing enterprises are usually supported by a heterogeneous mix of hardware and software, with many different types of database systems employed at any given time. When designing StockControlAgents so they may connect to such a variety of database systems it became apparent that some of the required features of these agents were particular to each database, whilst others were generic and could be applied to any StockControlAgent. In the initial stages of the implementation, the StockControlAgents had been using text files as their storage medium, modelled on MICROS records. Many new database systems no longer use text files however, so it was later decided to improve their capability to allow them to communicate with any ODBC enabled database. ODBC is an industry standard for database access. The work on this problem has yielded a common design that can be used as a base pattern

and applied to all StockControlAgents [Papaioannou99]. The DataQueryAgent is discussed later in Section 6.4.1.

6.3.5 ManufacturingAgents, MaterialsAgents, PurchasingAgents and DispatchAgents

These particular agent types have been classified as having secondary importance to this initial study. Currently all three are represented in the SOP systems by “dumb” static agents. By dumb we mean that they are merely communicative and possess no internal logic to perform any particular tasks. They are able to simply acknowledge communication from other agents, and represent a definite avenue for further investigation and research. However, their presence in the systems allows us to begin to explore the issues involved with multi-hop mobile agents vs the client/server paradigm.

6.4 Considering Lifecycle and Maintenance Issues

The implementations described in this thesis are proof-of-concept systems. They are used in our experimental work to demonstrate that real world software can be built with mobile code systems. In addition, we wished to measure the degree of flexibility, coupling and semantic alignment offered by the mobile code abstraction. Further, to fully consider the support provided for building real world systems we examine the full lifecycle phases of software systems. These include issues relating to design, implementation, runtime and maintenance. The resulting knowledge and supporting tools and are discussed in the next sections.

6.4.1 DataQueryAgent: A Proto-Pattern for Database Query

A major goal of the work described in this thesis has been to build agile software systems. For the software architectures implemented in this study to achieve this throughout their lifetimes, they must be capable of querying a variety of new or legacy databases. Investigation into this problem has generated an effective and reusable proto-pattern that can be used to build agent database query systems [Papaioannou98]. The DataQueryAgent, shown in Figure 25, can be decomposed into several constituent parts, which are described in the following sections.

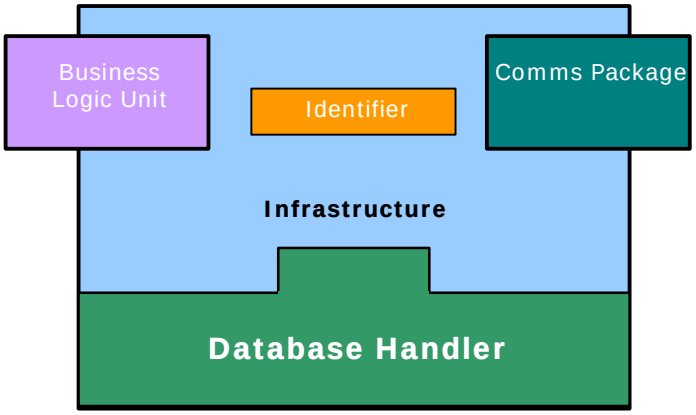


Figure 25. DataQueryAgent Architecture

6.4.1.1 The Infrastructure

The infrastructure provides the system creator with the facilities to communicate with, and manage the lifecycle of agents in the system. The environment in which the agent will execute normally dictates the infrastructural requirements, although they are usually accessible through the framework libraries or via class inheritance. For example, in our implementations these facilities are attained by extending the abstract `Aglet` class.

6.4.1.2 The Identifier

The Identifier plays an essential role in system security and traceability. Whilst it is more usual for mobile agents to carry an Identifier, static agents must also be able to prove their credentials. In future implementations, we imagine that StockControlAgents would be able to generate PurchaseOrderAgents and WorksOrderAgents in order to fulfil unsatisfied orders. Part of the parent's Identifier would be handed to these child agents, as proof of their origin on dispatch to another host.

6.4.1.3 The Communication Package

The Communication Package handles the incoming communication from querying agents and translates this into a format the Business Logic Unit or Database Handler components are able to understand. Inter-agent communication methods vary between different agent environments, as do the communication protocols and requirements of differing agent solutions. In some examples, simple `String` matching is sufficient for simple communication. However, interactions that are more

complex may require an attempt at semantic level communication. The use of Agent Communication Languages (ACL's) such as KQML [Labrou96] is typical of the more advanced approaches that are being proposed to solve these problems. To handle the requirement for a variety of communication methods, the Comms Package can be interchanged by the software designer with respect to their particular requirements.

6.4.1.4 Business Logic Unit

The Business Logic Unit is used to understand communication and queries from other agents, and generate a course of action to fulfil those requests. In the SOP scenario, when an OrderAgent is dispatched by the SalesAgent, it encapsulates an `Order` object. Upon arrival at the StockControlAgent, it will attempt to fulfil that order, a task that in itself can require some simple logic. For example, for simplicities sake an `Order` object only contains descriptions of the full products that are expected. Although the OrderAgent may only be aware that it requires one hundred widgets by Tuesday, the StockControlAgent may include some logic that translates this request into one where a widget must be supplied with a grommet and two nuggets. Thus, the Order actually requires one hundred widgets and grommets, plus two hundred nuggets. More probably, the StockControlAgent will query another database to retrieve the Bill of Materials for the product. Since all the OrderAgents will require this same logic, it is clear that including it as part of the DataQueryAgent is the best solution. By keeping the size of the `Order` and the encapsulated logic low, the size of the OrderAgent is kept small, reducing network traffic.

6.4.1.5 The Database Handler

The Database Handler deals with connecting to a database, retrieving information from it, updating it, or even switching databases transparently to the requesting agent. It works in tandem with the Business Logic Unit to fulfil the request of a querying agent. The Database Handler ensures that the DataQueryAgent is capable of interfacing with many different types of data source.

The examples shown in Figure 26 address a large percentage (but by no means all) of the real world situations and the methods currently being employed to query databases within a manufacturing enterprise. Connecting to a new type of database ostensibly requires only the production of a new Database Handler. However, we make no

claims about the ease of this task. It is understood that access to a database is not all that is required; there remains the difficult problems of understanding the schema used in the new database before specific information can be retrieved. Work towards this goal can be seen in the efforts of the EDI and STEP/PDES community.

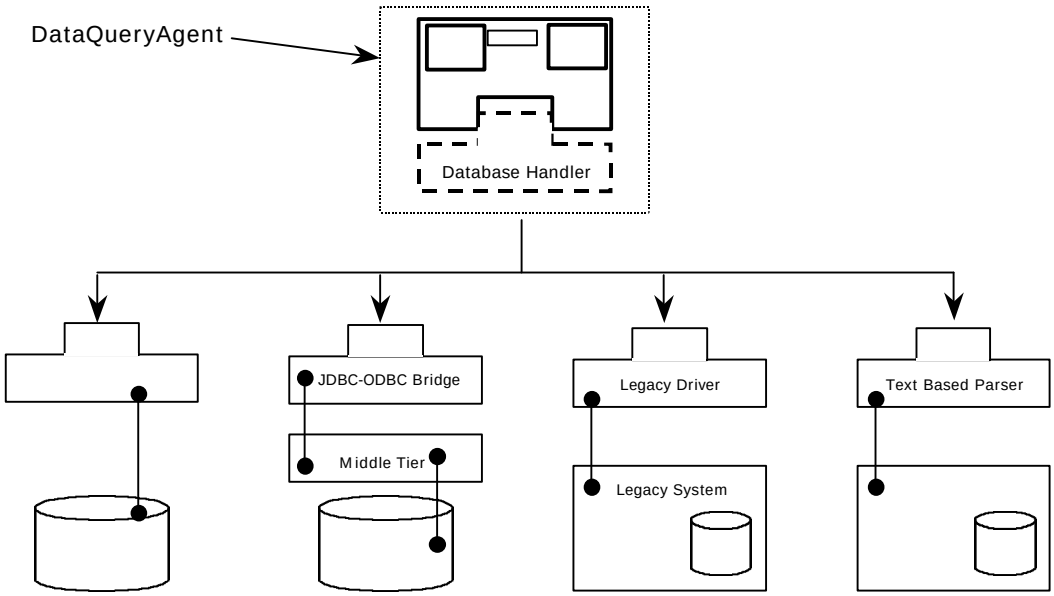


Figure 26. The DataQueryAgent with examples of different DataHandler modules

6.4.2 The Data Connector Tool

When constructing the StockControlAgents for our implementations, using the DataQueryAgent pattern, it became apparent that the most arduous task involved was in making the connection to a database. Whilst on the surface a relatively simple task, there are several variables that must be configured correctly, and a number of JDBC interfaces that must be used accurately. To alleviate the problems this caused, the DataConnector tool was produced to automate some of these tasks.

The DataConnector Tool is a Java program, with a user interface that allows the user to insert the required parameters for connection to a JDBC compliant data source. The validity of these parameters can be repeatedly tested, using the refresh, update and test facilities, until the correct configuration is achieved. Once a satisfactory connection has been made, this data is then exported by serialising it to disk. Each StockControlAgent can then be given a reference to the file that contains the particular information they require to connect to their specific database.

6.4.2.1 Benefits of DataConnector

The biggest advantage in using this tool is the ability to test connections to a database and server across the network, or even the Internet. If a virtual enterprise were to decide to use mobile agent technology as a tool for rapid integration, it is likely that one of the collaborators (or their systems administrator) will have some prior experience in using the technology. The DataConnector tool allows a single administrator to test all the required database connections between the relevant systems, and produce a set of connection information files that can be forwarded to the respective sites. Moreover, if the agent environments and servers have already been set up, a Messenger agent could deliver the files, and the DataHandlers could be completed and initialised automatically. The lightweight nature of a connection information file means that continued use of the agent system would allow an administrator to build up a set of predefined files for various configurations that would accelerate the speed with which new collaborators or data sources could be added in the future, increasing the system agility and responsiveness of the enterprise.

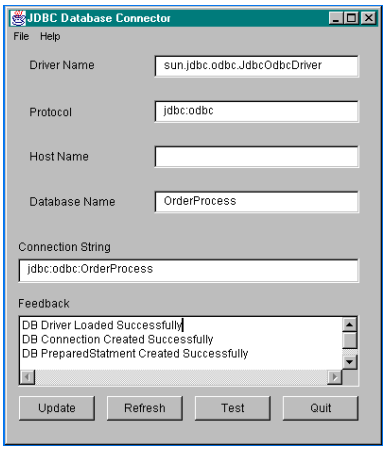


Fig 27 Screenshot of DataConnector

6.5 Concluding Remarks

In this chapter, we have described the realisation of our Sales Order Process model. We have produced two prototype implementations in order to evaluate the mobile object and mobile agent abstractions. The major processes identified in the overall business logic of the SOP have been embodied as agents in these systems, which comprise a mixture of static and mobile agents. Each individual type of agent created has been reviewed and discussed and their relationships examined.

The major difference between the two systems is the physical and conceptual location of the business logic associated with processing stock query results. In the mobile object version, this logic remains in the SalesAgent and is in an analogous position to where it would be found in a traditional client/server system. In the mobile agent

version, this logic is encapsulated within the mobile OrderAgent. In the former, the processing of the results must take place after all the data has been returned to the client, whilst in the latter the decision can be made locally to the data source by the mobile agent.

At the start of this chapter, we mentioned that part of the rational for this study was to demonstrate the feasibility of building real distributed systems with this new technology. We have accomplished that. We have built two prototype Sales Order Process software systems, based on a real world model, with mobile code technology. In addition, through consideration of the lifecycle and maintenance issues of these systems we have developed a proto-pattern to assist in the modular creation of DataQueryAgents. Supporting this pattern is a small tool, the DataConnector tool, which allows system administrators to rapidly connect DataQueryAgents to their data sources.

During the case study, described in Chapter 6 we also established several real world requirements for such systems. These have been identified as “scenarios for change” that can be used to evaluate how well each prototype responds to the types of pressures experience by real world software systems. The evaluation process and results are described in the next chapter.

7 Evaluation

7.1 Introduction

The previous chapter described the implementation of two mobile code systems. The rational for their construction was to evaluate the mobile object and mobile agent abstractions, in an attempt to understand exactly what each has to offer, and how that might affect how we build distributed systems. In this chapter, we evaluate how successfully each prototype responds to the *scenarios for change* that were generated from data collected in the case study of I.T.L, and report on the lessons learned and insights gained during these experiments.

7.2 Generating Useable Metrics

Evaluating software architectures is a notoriously hard task [Whitmire97]. There are very few established techniques or measurements for gathering data, and although software engineering as a discipline strives to emulate the classical sciences, we are still a long way off. Instead of formal equations, we have methodologies for developing metrics. They include: the Quality Function Deployment approach [Kogure83], the Software Quality Metrics approach [Boehm76] [McCall77] and the Goal Question Metric (GQM) approach [Basili94] [Solingen99]. Basili's GQM methodology was selected to evaluate the systems as it enjoys widespread popularity and support within the software engineering community.

In the next sections, we present an overview of the GQM methodology, and the principle goals, questions and metrics identified for the systems.

7.2.1 The Goal

The GQM methodology is based upon the assumption that to gain a practical measure one must first understand and specify the goals of the software being measured, and the goals of the measuring process. More specifically, it is important to specify what is being evaluated, what task it should fulfill and from what perspective to view the measurements. Once this framework has been established, it is possible to direct investigation and measurement towards the data that defines the goals operationally. The generated framework is also useful when interpreting the data.

The overall goal of our evaluation can be stated as:

> "To evaluate each prototype system from the industrialist's perspective, with respect to satisfying the industrial motivations to support system agility"
> (see section 5.5)

7.2.2 The Questions

Having stated the goal, the process is continued by generating a broad set of questions that may provide some indication of the individual issues encapsulated by the main goal. The objective is to generate as many questions as possible, including redundant or invalid questions. As the process continues, it is usual to develop a hierarchical set of questions that can subsequently be narrowed. This refined set can then be answered through tangible measurements made on the system.

To this end two workshops were held, one at MSI, Loughborough University, and one in the Computer Science Department of Reading University. In order to evaluate the prototypes with respect to the issues identified in section 5.5, the initial questions focused on system complexity (how easy is it to understand), and system agility (how easy is it to change). The results of these workshops were a large and varied set of questions, with many superfluous or duplicate entries. This is an expected part of the Basili methodology. Table 5 lists the focused set of questions that remained after refining.

7.2.3 The Metrics

After several iterations of refinement, and some healthy pruning, a set of usable software metrics remained that could be used to evaluate the two mobile code systems. These are shown in Table 6.

On their own, most of the generated metrics are extremely narrow in their focus. However, through combination, it is possible to arrive at some useful measures of a software system. In the following sections, we examine how these metrics can be used to evaluate the implemented systems, and discuss how well each prototype performs.

Generated Questions	Metric Number
How well does the system support change?	
How easy is it to understand the system?	
How many business entities map onto data abstractions	(1)
How many business processes map to software methods	(2)
Which real world entities that are mobile are also mobile in the system	(3)
Which real world entities that are static are also static in the system	(4)
How many components are there in the system	(5)
How many lines of code are there	(6)
How many comments are there	(7)
How easy it was to modify the system?	
How many conceptual entities must be changed - for example requirement a)	(8)
How many objects must be changed	(9)
How many src files must be changed	(10)
How many interactions must be changed	(11)
How many components are there in the system relative to the size	(5) + (6)
How many real world entities map to a software component	(1)+(2)+(3)+(4)
How many components must be changed	(9)
How many interactions must be changed	(11)
How many inter-entity connections are there	(12)
How many methods of the object are public	(13)

Table 5. Questions generated using the Basili GQM Method

Metric	Nature of metric
(1)	Identify information-based abstractions in the real world. Compare with info based abstractions in the software
(2)	Identify process-based abstractions in the real world. Compare with processes evident in the software.
(3)	Identify mobile elements of the real world, compare with mobile elements in the software
(4)	Identify static elements of the real world, compare with static elements in the software
(5)	Count the components
(6)	Count lines of code
(7)	Count comments, and get ratio of comments/method
(8)	Count num changes to entities for each requirement
(9)	Count num changes to objects for each requirement
(10)	Count num changes to interactions for each requirement
(11)	Count how many files are changed for each requirement
(12)	Count number of inter object method invocations
(13)	Count number of public methods

Table 6. Metrics Generated using the GQM Method

7.3 Evaluating Semantic Alignment

It has been demonstrated that semantic alignment between real world abstractions and components of a software system is important when attempting to build agile software systems [Coutts98b]. It is also a factor in how responsive a software system may be to change. To understand what the implications are for semantic alignment, when using mobile code, and to compare the two mobile code prototypes, we require some way of measuring how well the abstractions of the real world are embodied in software, and how well they resemble the real world model. For this, we have developed a term called Conceptual Diffusion.

7.3.1 Conceptual Diffusion

Conceptual Diffusion is defined as a measure of:

> "The degree to which a single concept or semantic abstraction in the application domain maps to the components in a software system."

Therefore, we may say that:

> ***CD* = A/B**

Where *CD* is conceptual diffusion, A is the number of concepts included in this abstraction, and B is the number of components in which this abstraction is embodied.

Conceptual diffusion can be examined at different levels of granularity to gain different perspectives on a situation. For example, in a software system that is intended to support a Sales Order Process we expect the concept of an Order to be present. On analysis, we find that in both the agent and the object systems the concept of an Order is split over four separate components. Thus, in these two systems, the concept of an Order can be said to have a conceptual diffusion rating of four (see Table 7).

Table 7 also shows the results of metrics (1) and (2). These metrics are examples of examining conceptual diffusion at a larger level of granularity. For example, metric (1) requires the identification of all the information-based concepts within the real world, and a comparison with their counterparts in the software systems. Since Order is an information-based abstraction, it is therefore included in the results of metric (1). We may use Conceptual Diffusion to gain an insight into how well concepts or abstractions are embodied in software.

Objects	Info Abstractions		Process Abstractions						SOP Logic	
	Order	Customer	SA	SCA	PC	M	P	D	MobAg	MobOb
BaseAglet			✓	✓		✓	✓	✓		
DBAglet				✓						
OrderAglet	✓								✓	✓
SlaveItin									✓	✓
SlaveDetails				✓						
SalesAglet			✓							✓
Result									✓	✓
GenericTask									✓	✓
StockCommit Task									✓	✓
DBStockRequest Task									✓	✓
NewOrderDialog			✓							
Order	✓									
OrderListEntry			✓							
OrderList			✓							
Product	✓									
ProductList				✓						
FutureLevels	✓									
OrderNumbers			✓							
SlaveList			✓							
Conceptual Diffusion	4	N/A	7	4	N/A	1	1	1	6	7

Table 7. Analysis of Conceptual Diffusion Present in Mobile Code

7.3.2 Semantic Alignment

Conceptual Diffusion in itself is a measure of how well a software system is semantically aligned with those business processes it is trying to support. As it stands however, the conceptual diffusion measure remains relatively fine grained in its perspective. It does not offer an overall view of a system, rather an insight into a particular abstraction.

To gain an overall perspective of a system, a compound metric has been devised. It is a combination of metrics (1) to (4) and is termed the Semantic Alignment Metric:

$$SA = \left\{ \frac{Is}{Ir}, \frac{Ps}{Pr}, \frac{Ms}{Mr}, \frac{Ss}{Sr} \right\}$$

where SA is semantic alignment, *I* is information based abstractions, *P* is process based abstractions, *M* is mobile components, *S* is static components, *s* denotes in software and *r* denotes in the real world. Thus, $\frac{Ps}{\mathrm{Pr}}$ is the ratio of process-based abstractions in the software to the process based abstractions in the real world.

Mobile elements	Mobile agent	Mobile object
Order	✓	✓
Products		
Materials		
Static elements	**Mobile agent**	**Mobile object**
Sales	✓	✓
Stock Control	✓	✓
Production Ctrl		
Manufacturing	✓	✓
Purchasing		
Dispatch	✓	✓

Table 8. Results of Metrics (3) and (4)

This metric can be used to analyse a system and to assess how well the software system reflects the semantics of the application domain. A comparison with the ideal alignment of {1,1,1,1} can be used as a measure to gauge how difficult it might be to understand the software, given an understanding of the application domain. Table 8 shows the results of metrics (3) and (4).

By combining the results of the first four metrics, we are able to state that:

For the Mobile Object System Semantic Alignment = {4,22/6,1/3,2/3}

For the Mobile Agent System Semantic Alignment = {4,21/6,1/3,2/3}

7.3.3 Commentary

The results of the Conceptual Diffusion and Semantic Alignment analysis show that both Mobile Agent and Mobile Object systems should be easy to understand, as the abstractions in the real world align reasonably well with the components of the software systems. The information abstractions from the real world are on average spread over four components in the implementations. When considering mobile and static component alignment, for both systems, a third of the components in the domain are modelled as mobile in the implementation, and two thirds of the static components in the domain are modelled as static elements in the implementations.

The difference in the two systems is shown when considering the semantic alignment of the business process. Here the mobile agent system is shown to have better semantic alignment than the mobile object system as the process logic for the SOP is contained *solely* within the OrderAgent and not diffused across both the SalesAgent and the OrderObject. Therefore, we can conclude that the mobile agent solution provides better semantic alignment with the real world business processes it supports.

If we consider contemporary distributed systems, we find they have no facility to support mobile components in a system. Therefore, they would be unable to implement any of the mobile abstractions. Instead, these abstractions would have to be diffused over several static components. If we consider the requirement for a stub, skeleton and IDL file, in addition to the client and server implementations, then the conceptual diffusion would be considerable. Since mobile code systems are equally adept at building static components, we can also postulate that mobile code systems

increase the semantic alignment between the real world and its supporting software systems, for any system that is not constructed from completely static components.

In addition, these new metrics are not merely restricted to use after the fact, but can be used proactively during the specification process, before any software has actually been built. Ensuring good semantic alignment of a software system before production will undoubtedly save both time and money in the long term. In particular, these metrics can be useful for identifying those components that should be mobile, and those that should be static. With increasing numbers of mobile code systems being built, this will prove an increasingly important aspect of system analysis and design

7.4 Evaluating System Agility

In order to evaluate the agility of a system it is necessary to make changes to that system. The case study of I.T.L. highlighted several real-world industrial requirements for agility that a company may have for a distributed SOP system. Using these requirements as scenarios for change, modifications to both the mobile agent and mobile object implementations were undertaken, in order to evaluate the agility of each system.

7.4.1 Change Capability

The GQM methodology enabled the derivation of several metrics that can be used to measure certain changes in a software system after modification. These measurements are specified by metrics (8), (9), (10) and (11). Individually, they enable us to measure narrow slices of change to a system. However, by combining these metrics it is possible to produce a more encompassing measure of agility. This set has been termed Change Capability, and is described by:

$$CC_{\acute{a} \rightarrow \hat{a}} = \left\{ \sum_{\acute{a}}^{\hat{a}} \ddot{a}o, \sum_{\acute{a}}^{\hat{a}} \ddot{a}s, \sum_{\acute{a}}^{\hat{a}} \ddot{a}i, \sum_{\acute{a}}^{\hat{a}} \ddot{a}\mathring{a} \right\}$$

where Change Capability CC, for a required change, is the set of the changes to the number of objects (*o*), the number of src files (*s*), the number of interactions (*é*) and the number of conceptual entities (å), between states á and â. A conceptual entity is

analogous to the abstraction or concept referred to in the previous sections. For example, it could be an Order, or a StockControlAgent. Interactions are those exchanges of information between objects, usually via method invocations, although for agents this also applies to any messaging dialogue they might enter. Changes to those interactions will usually imply changing a method signature.

Change Capability can be used to compare systems or to get a measure of the agility of the system relative to the ideal {0,0,0,0}. For the mobile object and mobile agent systems Change Capability for each requirement is summarised in Table 9.

Industrial Requirement	System	
	Mobile Agent	Mobile Object
The addition of new sales agents	{0,0,0,0}	{0,0,0,0}
The addition of new stock control centres	{3,3,1,2}	{3,3,1,2}
The removal of new additions	As A or B	As A or B
Allowing changes to the business logic of the SOP to be made easily	{1,1,0,1}	{2,2,0,2}

Table 9. Change Capability metric sets after "scenarios for change"

7.4.2 Commentary

Again, these results show that both systems are relatively easy to change. Adding new sales facilities requires only the instantiation of new SalesAgents that incurs zero changes to the system code. New stock control centres can be added through a low number of changes that are the same for both systems. The difference between the systems becomes apparent when making changes to the Sales Order Process logic. In the mobile agent system, this logic is contained *solely* in the single mobile OrderAgent, whereas in the mobile object system it is contained in both the SalesAgent and the OrderObject.

The Change Capability metric can be used by a system designer to evaluate how responsive to change their system has been after a specific change. It is possible to

deduce areas that require refactoring, or are particularly troublesome when undertaking change. For example, consider the CC set {5, 20, 20, 1}. We see that for this change, although only one conceptual entity was changed, there were twenty changes to source files, five changes to objects, and twenty changes to the interactions of those objects. Changing the signature of twenty methods in five objects to enable a change in a single entity can cause serious problems and should lead the designer to review how diffuse this particular entity actually was. Of course, this is also revealed by the Conceptual Diffusion metric.

While both implementations have demonstrated they are relatively agile, the question of whether they are more agile than a contemporary distributed system remains open. Certainly, it is unlikely that a traditional system will be any more agile than the mobile object system, since Remote Computation and Client/Server are very close in terms of the abstraction they offer. Nevertheless, we are able to assert that the mobile agent system has shown that it is more agile than the mobile object system. This increased agility was due to the reduced conceptual diffusion and improved semantic alignment that the mobile agent abstraction allows. In the next section, we pursue this matter by examining loose coupling, a central issue to building agile software systems.

7.5 Evaluating Loose Coupling

To build loosely coupled systems, components of that system should not be linked directly to form a complex network of interactions and inter-dependencies. Instead, they should remain distinct abstractions, embodying the concept of their real world equivalents. Components can then be assembled into a software system, with no prior knowledge of each other.

7.5.1 Evaluating Coupling in Mobile Code Systems

We have already seen in the preceding sections that distributed systems built with mobile code are able to minimise conceptual diffusion. This enables an extremely good alignment between real world processes and their supporting software counterparts. On examination of the static software entities in our systems, for example SalesAgents, StockControlAgents, ManufacturingAgents, etc, we find that

they are fully decoupled from each other. During execution of the system, there is no communication or interaction between any of the static components. Any communication that does take place within the systems is between static and mobile entities. Until a mobile entity alights at a host and attempts to interact with a static one, there is no coupling between any of the components. This is significant, since the system only experiences tighter coupling during a dialogue between components, i.e. when a mobile entity wishes to communicate with a static one. Of course, this dialogue depends upon prior knowledge on the part of the mobile entity as to what language the other agent understands, be it a syntactic dialect, or a more complex semantic conversation. In a private, controlled system however, this knowledge will always be available. In addition, since there are very few types of component that are mobile it is simple to alter the interactions, by updating the mobile agent population. Research is being undertaken so a dialogue may be established with no foreknowledge [Martin99]. Although this is currently in the static, intelligent agents domain, in time it will naturally be applied to that of mobile agents.

7.5.2 Commentary

Our prototype systems have demonstrated extremely low, if not non-existent, component coupling until runtime. Contemporary distributed systems such as CORBA do support loose coupling in the same inherent manner [Coutts98b]. Components in these systems that wish to communicate require implicit knowledge of each other's interfaces. These interfaces are the central aspect of building distributed systems with traditional technology.

> "You should be able to look only at the IDL and know precisely how to implement against it." [Vinoski99]

Therefore, even if the key conceptual abstractions remain embodied in large grained components, for these components to interact they must be aware of each other *a priory*, and inevitably end up intermeshed with each other. The work of Coutts and Edwards has shown that it is possible to build loosely coupled systems with traditional technology by employing additional design patterns and forethought. The author believes that being required to follow this enforced route is simply increasing the cognitive complexity of building distributed systems. Something that is already an onerous task.

This circumstance arises since location transparency, the abstraction employed in contemporary distributed systems, does not support loose coupling inherently. Distributed systems built with this abstraction rely on component interface signatures for identification, and to facilitate communication. Coutts and Edwards [Coutts98b] have demonstrated that with further software architectures a certain degree of loose coupling can be achieved. Their use of the Mediator pattern has one drawback however – all components that wish to interact must do so via the Mediator. The strength of this approach is also its main weakness. By enforcing a policy of mediation, the distributed system is also subjected to centralised control, and thus the Mediator is a single point of failure. Building distributed software systems with a single point of failure is known as a bad technique.

In a contemporary distributed system the concept of physical location is hidden. However, for two components to interact there must be some form of identification involved. This identification manifests itself through the interface types of the interacting components. Therefore, in reality the purpose of identification by interface is to enable the location of a component that can provide the required services. The core information in the task of locating a component is no longer physical location, rather it is the interface. Although the major tenet of this abstraction is location transparency, it is clear that the task of locating components remains. It has merely been replaced by an alternative method. Of course, practitioners of contemporary distributed systems argue that location transparency as provided by the abstraction is for the benefit of those who build and use the system. This may be the case, but we must also consider the implications of using this abstraction on the supporting technology, i.e. the distribution infrastructure.

Distributed System Technology	Locator Requirement	Dialogue Requirement
Traditional Technology	Interface	Interface
Mobile code systems	Location	Interface

Table 10. Requirement of Distributed Systems

On the other hand, components in distributed systems built with the local interaction abstraction do not rely on interface signatures to be located. Instead, they employ physical location as the information required for location. This is an important difference. By retaining location as the locator, the mobile code abstraction divorces the distribution mechanism from the dialogue constraints. This is shown in Table 10.

This separation has important implications for how tightly coupled a system might be. By divorcing distribution from dialogue, distributed systems can be much more loosely coupled until runtime. At the outset, all that two components who wish to communicate must know about each other is their respective locations. It is only when they actually wish to interact that they become more tightly coupled. The difference to contemporary technologies is in the timing of when it is required.

The implications of this subtle change are fundamental. System agility is affected by the coupling of components within a system, and in this respect, we argue that local interaction does indeed support looser coupling than traditional distribution technologies. By divorcing the mechanism for distribution from the dialogue, components in a system can be loosely coupled right up until the moment of interaction. Although once engaged in dialogue the components become tightly coupled, the moment of coupling has been delayed. Therefore, we may conclude that mobile code systems are more loosely coupled, and this looser coupling enables improved system agility when compared with traditional distribution technology.

The important issue to understand is why there are such marked differences between the abstraction offered by current distribution technologies and that offered by mobile code. In chapter one we examined the history of computing and saw how the computing landscape we inhabit today has been formed through the gradual layering of ascending abstractions. This is not a problem, since abstractions are an extremely useful tool for reducing the complexity of a situation, removing the minutiae so one might contemplate the problem at hand with clarity. However, what is important about abstraction is the importance of using an appropriate one. One that is able to accurately describe the real situation, without losing any important information.

It has been the author’s belief that the major tenet of RM-ODP systems, that of location transparency, is fundamentally flawed in this respect. The first notion of this

abstraction arose when Birrel and Nelson attempted to take the extremely successful abstraction of IPC, and apply it to many networked machines, in order to make local and remote calls look identical. This philosophy has prevailed and been extended so that we currently employ an abstraction that attempts to make every object or component in a distributed system believe they are executing in the same computing machine. However, by attempting to "shoehorn" an abstraction that was perfectly suited for the underlying hardware, i.e. a single von Neumann machine, onto many computing machines an important piece of information has been lost from the abstraction – location. Waldo et al identify several problems of distributed systems but do not offer a clear reason for these problems. We propose that it is due to the loss of location from the distribution abstraction. Identification of components in the network can no longer be achieved via their location, instead they must be identified by their interface signatures.

The assertion of the author is that although this technology can indeed build successful distributed systems, the drawbacks do not warrant the effort. The price for using the interface as a locator is tightly coupled systems that are difficult to change. Instead of enabling location transparency, mobile code systems enable local interaction, an abstraction ideally suited to single von Neumann machines. By using physical location as a locator, mobile code systems are able to separate the issues of distribution from the issues of dialogue, and thus these systems are more loosely coupled. Additionally, they provide improved semantic alignment, and thus reduce the cognitive complexity of the system.

Employing the correct abstraction can have fundamental consequences to building distributed systems. Instead of a flat plane of components that all believe they are in the same host, the mobile code abstraction removes this opacity of RM-ODP and exposes the rich network environment.

7.6 Concluding Remarks

Evaluating software systems is never an easy task. The evaluation in this thesis has been undertaken following Basili's GQM methodology. Using this technique a set of tangible metrics was developed to assist in the evaluation of the two mobile code systems. The motivation for the experimental work carried out in this thesis was to

demonstrate the feasibility of actually building distributed systems with mobile code technology, and to investigate the implications for system agility when using this new paradigm.

We initially examined the issue of semantic alignment and compared our two prototype systems. The experimental work has shown that by reducing the conceptual diffusion in a system, the mobile agent abstraction is able to offer improved semantic alignment with the business process it is intended to support when compared to the mobile object system. The difference is barely significant in our systems, but could easily be magnified in a full size system. In the process of this evaluation, two software metrics have been developed to assist the system designer in identifying which components, if any should be mobile.

On examination, system agility is a harder issue to resolve. The experimental work has shown that mobile code systems are relatively agile, with the mobile agent abstraction being slightly more so than the mobile object abstraction. The differences in each implementation with respect to agility are identical to the differences in semantic alignment. This is due to lower conceptual diffusion in the mobile agent system, something that is enabled by the autonomy of the agent metaphor.

When looking at loose coupling we see no difference between the mobile object and mobile agent prototypes. However, in general component coupling in these systems is extremely low. This is in marked contrast to distributed systems built with the location transparency abstraction. Although our work does not shed any further quantitative light onto this matter, our observations do support the argument made in Part I of this thesis: that location transparency is fundamentally flawed. Our conclusion is that this is further exacerbated by combining the information used for location of components with that required for a dialogue. Local transparency on the other hand separates these two issues, and is thus able to build more loosely coupled systems that are more responsive to change.

8 Conclusions

Building distributed systems is not a new endeavour. We have been doing so for as long as we have been networking computers. However, the types of system being built, and the nature of the underlying network are evolving beyond the wildest dreams of the early network pioneers. Networks are becoming pervasive in society, and the dream of ubiquitous computing is finally being realised. These new networks bring new requirements for how we build distributed systems. We can no longer guarantee network reliability or even topology. Our existing technologies and infrastructures are beginning to creak under the strain.

This thesis has been concerned with how we build distributed systems. Instead of focusing merely on the technology used to implement them, we have also focused on the abstractions employed in their construction. These immensely powerful concepts allow us to manage the complexity of a situation, by removing those details we consider inessential. After all, the central essence of any paradigm is the abstractions it embodies. The major contributions of this thesis have been:

- An extensive philosophical argument and critique of abstractions for distribution
- The demonstration of the feasibility of building real-world distributed systems with mobile code infrastructures
- The creation of the new software metrics of Conceptual Diffusion, Semantic Alignment and Change Capability
- Quantitative comparisons of the Mobile Agent and Remote Computation abstractions

In Part I, *Understanding*, we traced the emergence of abstractions in computing, and built a philosophical understanding and critique of the abstractions used to construct distributed software systems. The central thesis of this work is that by employing the location transparency abstraction, and attempting to create the illusion that all components exist within the same computational machine, contemporary distributed systems are fundamentally flawed as they break the Tower of Abstractions by attempting to impose an unsuitable abstraction on the underlying computational substrate. We have demonstrated that location transparency was a wrong fork in the evolutionary road of distribution. Our proposal is that a new abstraction, local interaction (embodied in mobile code infrastructures), that returns to the core

successes of the von Neumann computational machine is a more suitable abstraction with which to build distributed systems in today's ubiquitous networks. Removing location from the abstraction has proven detrimental to the agility of systems built with this technology, since the issues of distribution have become tied with those of dialogue. Whilst we advocate the use of abstraction, we believe that location transparency loses essential information when employed. We believe that Part I of this thesis contributes by raising the level of conceptual understanding surrounding the mobile code paradigm.

The arguments presented in Part I are extensive, and a full experimental investigation was deemed beyond the scope and timescale of a PhD. Instead, our horizons were shortened to encompass the first steps along the long path of validating the argument. Part II, *Using and Evaluating,* is therefore a report on our experiences of mobile code in the real world. To date, the mobile code research arena has remained relatively immature, and the dearth of real systems has hampered its development. With this in mind, our experimental work was based upon a business process model generated from an industrial case study. We reported on the creation of two prototype systems that embodied the Mobile Agent and Remote Computation abstractions, part of the mobile code family of abstractions. In this, we have achieved our first aim; to demonstrate the feasibility of building real world distributed systems with mobile code. We also wish to comment on the relative merits of each prototype.

In the course of the experimental work, we subjected our systems to real world pressures in the form of Scenarios for Change, also generated from the case study. During the subsequent evaluation, we developed several metrics using the Basili GQM methodology. The metrics of Conceptual Diffusion, Semantic Alignment and Change Capability have proved to be useful techniques for evaluation that can be used during both the specification process, and post construction. In addition, we have tried to consider the full lifecycle of our systems, an exercise that has produced several supporting tools and proto-patterns.

Our evaluation of the two mobile code prototypes draws us to conclude that the mobile agent abstraction is the more useful to employ. From our experiments, we observe that mobile agents enjoy increased semantic alignment and system agility when compared to the remote computation abstraction. The differences in each

implementation arise due to the lower conceptual diffusion of the mobile agent system, something that is enabled by the autonomy of the agent metaphor.

We believe that this thesis is a beginning, an initial monograph on abstractions for distribution. It is clear that location transparency is unsuitable for some types of system we wish to build, and that mobile code offers a viable alternative. This is not to say that all distributed systems should be built with mobile code. Mobile agents offer us a solution for networks where topology, quality of service and varying bandwidth are the core issues. We should appreciate the nuances of each abstraction, so that we may apply them in the correct situation.

8.1 Future work

As has been mentioned, the arguments made in Part I are extensive, and their scope beyond that which can be considered in the timescale of a PhD. This is not to say we have not contemplated what would be required. The experiments described in this thesis have been a first step. We have demonstrated the viability of mobile code, and our results indicate that the mobile agent abstraction supports good system agility. The question of whether mobile code technology is superior to contemporary technology remains open. It is very difficult to compare the two, since the maturity levels of the technologies differs greatly. Distributed systems built around the RM-ODP model have been around for over a decade with much industry development, whilst mobile agent systems have been around merely a few years.

We believe the next stage of validation for our philosophical argument would be to undertake a course of research to directly compare Mobile Agents with RM-ODP. To avoid the differences in technology maturity, we envisage building each abstraction from the ground up. A clean room implementation of both abstractions would allow a more valid and comprehensive comparative analysis. Further, it is clear that software patterns and software metrics evolve throughout their lifetime. Through use, practitioners are able to refine them. We believe additional software metrics would support this investigative work.

As has already been mentioned, an obvious avenue for future work would be to continue the SOP implementation undertaken in this thesis. The current model

embodied in our prototypes has many areas where it can be expanded. Increasing the size and complexity of our systems would allow us to reapply the scenarios for change. A comparative study with our current results would be a valuable exercise to ascertain how much of an effect size and complexity has on system agility. We should also be searching for collaborative partners on other continents to truly test how successfully each system supports distribution.

Finally, the creation of a modelling language that includes the facility to specify mobile components would be an invaluable addition to the system designer's toolbox. Current modelling languages, such as UML [Booch97], do not include the concept of mobility. Extending *de facto* industry methodologies is a sure fire way to ensure widespread adoption of new ideas and technologies.

8.2 Commentary

Using mobility is not just about what the technology can do for you. It is also about a fundamental change of mindset. By removing the conceptual block that is the *plane of transparency*, distributed systems designers can begin to appreciate the rich environment that is the network. If we remain faithful to the Tower of Abstractions, and employ the network as our communications infrastructure, we draw on the strengths of the von Neumann machine *and* the network suite, whilst divorcing the issues of distribution from those of dialogue.

In hindsight, it is easy to illustrate the reasons our computing evolution meandered down the location transparency fork. Recently an expanding community has realised there are problems with this approach. As a software engineering community in the large, we must be brave enough to face up to those problems, and admit our mistakes. It is better to attack the problem as early as possible, than build ever more elaborate software constructs to support a dying abstraction. The ideas generated during the work undertaken in this thesis have allowed the author to view distribution from a different perspective. Local interaction is beginning to establish itself as a valid tool for building earthbound distributed systems, but it has already been considered for perhaps the ultimate distributed system - a space based network [Papaioannou99c]. There can be no question of location transparency being employed when the distances involved in this type of network are considered!

Mobile agents have shown considerable early promise. The future they depict is one of a rich network environment, inhabited by an ecology of autonomous agents. Nodes in the network become islands of resources, on which agents may alight to take advantage of resources locally. The population consists of mobile and static agents, all enjoying some level of autonomy, ranging from simple task specific instructions, to complex autonomous agent architectures. The mobile agents live in the network, able to migrate, clone, sleep, wake, but in reality insert a new layer of abstraction over the underlying computation substrate. They act for other agents, or their human owners. The static agents are brokers for immovable resources such as printers or databases. In this virtual ecology, we see the glimpses of our future computing.

List of Publications

Clements, P.E., Papaioannou, T. and Edwards, J.M., "Aglets: Enabling the Virtual Enterprise", Proceedings of the 1st International Conference on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement, ME-SELA '97, Wright, Rudolph, Hanna, Gillingwater and Burns (eds), Mechanical Engineering Publications, Loughborough University, July 1997,pp 425-432, ISBN 1-86058-066-1

Papaioannou, T., Edwards, J.M., “Mobile Agent Technology Enabling the Virtual Enterprise: A Pattern for Database Query”, in notes of Agent Based Manufacturing Workshop, part of the International Technical Conference Autonomous Agents '98.

Papaioannou, T., Edwards, J.M., “Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems”, International Journal of Robotics and Autonomous Systems, 27, pp 45-57, 1999.

Papaioannou, T., Edwards, J.M., “Mobile Agent Technology in Support of Sales Order Processing in the Virtual Enterprise”, in [Camarinha-Matos et al]

Papaioannou, T., “Mobile Agents: Are They Useful for Establishing a Virtual Presence in Space?”, in notes of Adjustable Autonomy Symposium, part of the AAAI Spring Symposium Series, Stanford University, 1999.

Papaioannou, T., Minar, N., “Mobile Agents in the Context of Competition and Cooperation”, Proc. of MAC3 workshop, part of Autonomous Agents ‘99 conference, Seattle, 1999.

Papaioannou, T., Edwards, J.M., “Manufacturing Systems Integration and Agility: Can Mobile Agents Help?”, accepted for publication in Journal of Integrated Computers-Aided Engineering, IOS Press. To appear in January 2001 Issue.

Papaioannou, T., Edwards, J.M., “Towards Understanding and Evaluating Mobile Code Systems”, accepted for publication in Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers. To appear in 2000.

References

Abadi96 Abadi, M., and Cardelli, L., “A Theory of Objects”, Monographs in Computer Science, Springer-Verlag, Berlin, 1996.

Accetta86 Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M., “MACH: A New Kernel Foundation for UNIX Development”, Proc. Summer USENIX Conference, pp 93-112, 1986.

Adobe85 Adobe Systems Inc., “The Postscript Language Reference Manual”, Addison-Wesley, 1985.

Agha97 Agha, G., “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems”, in Najm, E. and Stefani, J.B., Eds, “Formal Methods for Open Object-based Distributed Systems”, Chapman & Hall, 1997

Andrews82 Andrews, G.R., “The distributed programming language SR – mechanisms, design and implementation”, Software Practice and Experience, Vol 12, pp 719-753, 1982

Andrews83 Andrews, G., Schneider, F., “Concepts and Notations for Concurrent Programming”, ACM Computing Surveys, 15, pp 3-43.

Apple92 Apple Computers, “Dylan, an Object Oriented Dynamic Language”, Apple, Cupertino, CA, 1992.

Arnold99 Arnold, K., Wollrath, A., O’Sullivan, B., Sheifler, R., Waldo, J., “The Jini Specification”, Addison-Wesley, 1999.

Backus78 Backus, J., "Can Programming be Liberated from the Von Neumann Style?”, Comm. ACM 21 (8), pp. 613-641.

Ball98 Ball, K., McClain, D., Minium, D., 1997, "Enterprise Enablement for Java Applications", XDB SystemsReferences

Barber98 Barber, M., Weston, R., "BPR Scoping Paper", IJPR, 1998.

Barlow97 Interview with the Managing Director of I.T.L., Mr David Barlow, 1997.

Basili94 Basili, V.R., Caldiera, G., Rombach, H.D., (1994), “The Goal Question Metric Approach”, Encyclopedia of Software Engineering, pp 528-532, Wiley and Sons.

Baumann97 Baumann, J., Hohl, F., Rothermel, K., “Mole – Concepts of a Mobile Agent System”, Technical Report No 1997/15, Faculty of Computer Science, Stuttgart, Germany, 1997.

Ben-Ari90 Ben-Ari, M., “Principles of Concurrent and Distributed Programming”, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Bennet94 Bennett K.H., Ward M.P., 'Using Formal Transformations for the Reverse Engineering of Real-time Safety Critical Software' Proc. Second Safety-Critical Systems Symposium, Birmingham, 1994, pub. Springer-Verlag, ISBN 0-387-19859-8, pp. 204 –223

Berners-Lee92 Berners-Lee, T.J., Cailliau, R., Groff, J.-F., Pollerman, B., "World-Wide Web: The Information Universe.", in *Electronic Networking: Research, Applications and Policy*, Vol 2 (1), pp 52-58, Westport CT: Meckler Publishing.

Berners-Lee92b Berners-Lee, T., Fielding, R., Masinter, L., "Uniform Resource Identifiers (URI): Generic Syntax", available at http://www.ietf.org/rfc/rfc2396.txt

Birrel84 Birrel, A.D., Nelson, B.J., "Implementing remote procedure calls", ACM Transactions on Computer Systems, Vol 2, pp 39-59, 1984

Birtwistle73 Birtwistle, M. G., Dahl, O. J., Myhraug, B., Nygaard, K., "Simula Begin", Petrocelli/Charter, New York, 1973.

Blair91 Blair, G.S., et al. "Object-Oriented Languages, Systems and Applications", Pitman, London UK, 1991, cited in [Coutts98]

Bobrow88 Bobrow, D.G., De Michiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G, and Moon, D.A., "Common LISP object system specification", ACM SIGPLAN Notices, 23, September, 1988.

Boehm76 Boehm, W., Brown, J.R., Lipow, M., "Quantitative Evaluation of Software Quality", Proc. 2nd International Conference on Software Engineering, 1976, pp 592-605.

Boggs73 Boggs, J.K., "IBM Remote Job Entry Facility: Generalised Subsystem Remote Job Entry Facility", IBM Technical Disclosure Bulletin, 752, August 1973.

Booch94 Booch, G., "Object Oriented Analysis and Design with Applications", Redwood City, CA: Benjamin/Cummings, 1994

Booch97 Booch, G., Rumbaugh, J., Jacobsen, I., "Unified Modelling Language Semantics and Notation Guide 1.0", Rational Rose Software Corporation, CA, 1997.

Brener87 Brenner, J.B., "Open distributed processing", ICL Technical Journal, Vol. 5 (4), pp 613-637, 1987

Brooks95 Brooks, F.P. Jr, "The Mythical Man-Month: Essays on Software Engineering", Addison-Wesley, Reading, MA, 1995.

Burks46 Burks, A.W., Goldstine, H.H., von Neumann, J., “Preliminary Discussion of the logical Design of an Electronic Computing Instrument”, U.S. Army Ordinance Dept. Report, 1946.

Callear94 Callear, D., “Prolog Programming for Students”, Ashford Colour Press, England, 1994.

Camarinha-Matos98 Camarinha-Matos, L. M., Vieira, W., “Using Multiagent Systems and the Internet in Care Services for the Ageing Society”, appearing in [Camarinha-Matos et al], 1998.

Camarinha-Matos et al Camarinha-Matos, L. M., Afsarmanesh, H., Marik, V., eds. "Intelligent Systems for Manufacturing: Multi-Agent Systems and Virtual Organisations", Kluwer Academic Publishers, 1998, ISBN 0-412-84670-5

Cardelli85 Cardelli, L., and Wegner, P., “On understanding types, data abstraction, and polymorphism.”, ACM Computing Surveys, 17 (4), pp 471-522, 1985.

Carrot97 Carrot, A.J., Wright, C.D., West, A.A., Harrison, R., "Creating a distributed object-oriented integration framework for machine design and control ”, First International Conference on Managing Enterprises-Stakeholders, Engineering, Logistics & Achievement (ME-SELA ‘97) at Loughborough University, 22-24 July 1997.

Carver91 Carver, G. P., Bloom, H.M., "Concurrent Engineering through Product Data Standards", U.S. Department of Commerce, May 1991.

Carzaniga97 Carzaniga, A., Picco, G.P., Vigna, G., “Designing Distributed applications with Mobile Code Paradigms”, Proc. 19th International Conf. On Software Engineering (ICSE’97), 1997, Taylor, R., Ed., ACM Press, pp 22-32.

Cashin80 Cashin, P.M., “Inter-Process communication”, Bell-Northern Research Report, May 1980.

Cerf74 Cerf, V. and Kahn, R., “A protocol for Packet Network Interconnection”, IEEE Trans. on Communication, Vol. COM-22, pp 637-648, 1974.

Cerutti83 Cerutti, D., Pierson, D., “Distributed computing environments”, McGraw-Hill, 1993

Cheong83 Cheong, V.E., “Local Area Networks”, Wiley and Sons, 1983.

Chess97 Chess, D., Harrison, C., Kershenbaum, A. "Mobile Agents: Are They A Good Idea ?", in "Mobile Object Systems, Towards one programmable Internet", Edited by Vitek, J., Tschudin, C., Springer-Verlag Lecture Notes in Computer Science 1222, 1997, ISBN-3-540-62852-5.

Chomsky59 Chomsky, N., “On Certain Formal Properties of Grammers”, Information and Control, 2 (2), pp 137-167, 1959, cited in [Coutts98]

Church41 Church, A., "The calculi of lambda conversion.", Annals of Mathematics Studies, 6, Princeton University Press, Princeton NJ, 1941.

Clements97 Clements, P.E., Papaioannou, T. and Edwards, J.M., "Aglets: Enabling the Virtual Enterprise", Proceedings of the 1st International Conference on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement, ME-SELA '97 , Wright, Rudolph, Hanna, Gillingwater and Burns (eds), Mechanical Engineering Publications, Loughborough University, July 1997, pp 425-432, ISBN 1-86058-066-1.

Clocksin87 Clocksin, W.F., Mellish, C.S., “Programming in Prolog”, 3rd edition, Springer-Verlag, 1987.

Comer91 Comer, D., “Internetworking with TCP/IP Volume I: Principles, Protocols, and Architectures”, 2nd Edition, Prentice Hall, 1991

Coulouris94 Coulouris, G., Dollimore, J., Kindberg, T., “Distributed Systems: Concepts and Design (2nd Edition)”, Addison-Wesley, 1994

Coutts98 Coutts, I., A., "An Infrastructure to Support the Implementation of Distributed Software Systems", doctoral thesis (to be published), Loughborough University, 2001.

Coutts98b Coutts, I.A., Edwards, J.M., “Support for Component Based Systems: Can Contemporary Technology Cope?”, in [Camarinha-Matos et al], 1998.

Cox87 Cox, B. J., "Object Oriented Programming - An Evolutionary Approach", Addison-Wesley, Wokingham, UK, 1987, cited in [Coutts98]

Cox98 Cox, B. J., Opinion expressed in private correspondence via email, 1998.

Crichlow88 Crichlow, J.M., “An Introduction to Distributed Parallel Computing”, Prentice-Hall, 1988.

Cypser78 Cypser, R., “Communications Architectures for Distributed Systems”, Addison-Wesley, 1978.

DeMarco78 T. DeMarco, *Structured Analysis and System Specification,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978

DeRemer76 DeRemer, F., Kron, H.K., “Programming in the Large Versus Programming in the Small”, IEEE Transactions on Software Engineering, SE-2 (2), pp 80-86, 1976.

Dijkstra68 Dijkstra, E.W., "Goto Statement Considered Harmful", Comm. ACM, 24, pp. 147-148, 1968.

DoD61 Department of Defense, "COBOL, Revised Specification for a Common Business Oriented Language", 196.

DoD80 Department of Defense, "Ada Programming Language", Report MIS-STD-1815, Washington D.C., 1980.

DoD80b USA Department of Defence, “Reference Manual for the Ada Programming Language”, Proposed Standard Document, 1980.

Einstein39 Einstein, A., Infeld, L., “The Evolution of Physics”, 2nd Edition, Simon and Schuster, NY, 1960.

Franklin96 Franklin, S and Graesser, A., 1996, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", Proceedings of the 3rd Int. Workshop on Agent Theories, Architectures, and Languages, Published as Intelligent Agents III Springer-Verlag , Berlin, 1997, pp 21-35.

Fukuoka82 Fukuoka, H., “Interprocess communication facilities for distributed systems: a taxonomy and a survey”, Research Report, Georgia Institute of Technology, GIT-ICS-82/06, 1982.

Gascoigne94 Gascoigne, J.D., “CIM-BIOSYS Integrated System Implementation Toolset”, MSI Research Institute, Loughborough University, England, 1994.

Gershenfeld99 Gershenfeld, N., “When Things Start to Think”, Henry Holt & Company, 1999, ISBN 0805058745. in [Minar99]

Geschke77 Geschke, C. M., Morris, J. H. Jr., Satterthwaite, E. H., "Early experiences of Mesa", Comm. ACM, 20 (8), pp. 540-553, 1977.

Ghezzi98 Ghezzi, C., Jazayeri, M., “Programming Language Concepts”, 3rd ed., Wiley and Sons, 1998.

Glass99 Keynote speech given by Graham Glass, CTO of ObjectSpace at Autonomous Agents 99, Seattle, May 1999.

GoF93 Gamma, E., R. Helm, R. Johnson, & J. Vlissides, ``Design Patterns: Abstraction and Reuse of Object-Oriented Designs", *Proceedings, ECOOP '93*, Springer-Verlag, 1993.

Goldberg83 Goldberg, A., Robson, D., “Smalltalk-80: the Language and Its Implementation”, Addison-Wesley, Reading, MA, 1983.

Goldman95 Goldman, S.L., Nagel, R.N., Preiss, K., “Agile Competitors and Virtual Organisations”, Van Nostrand Reihold Publishing, 1995.

Gong99 Gong, L., “Inside Java 2 Platform Security: Architecture, API Design, and Implementation”, Addison-Wesley, 1999. ISBN: 0201310007

Goodenough75 Goodenough, J.B., “Exception handling: Issues and proposed notitation”, Comm. ACM, 16 (12), pp 683-696, Dec. 1975.

Gosling96 Gosling, J., Joy, B., Steele, G., "The Java Language Specification", Addison-Wesley, Reading, MA, 1996.

Gray83 Gray, J.P., Hansen, P.J., Homan, P., Lerner, M.A., Pozefsky, M., “Advanced program-to-program communication in SNA”, IBM Systems Journal, Vol. 22 (4), pp 298-318, 1983.

Gray97 Gray, R., “*Agent Tcl: A flexible and secure mobile-agent system*”, PhD thesis, Dept. of Comp Sci, Dartmouth College, June 1997.

Green80 Green, P.E. Jr, “An Introduction to Network Architectures and Protocols”, in [IEEE80].

Hammer93 Hammer, M., Champy, J., “Re-engineering the corporation”, Nicholas Braedly Publishing, London, 1993.

Harel87 Harel, D., “The science of computing: exploring the nature and power of algorithms, Addison-Wesley, USA, 1987.

Harel93 Harel, D., “Algorithmics: The Sprit of Computing 2nd Editions”, Addison-Wesley, 1993.

Hoare72 Hoare, C.A.R., “Notes on data structuring”, Structured Programming, Academic Press, pp 83-174, 1972

Hoare74 Hoare, C.A.R., “Monitors: An operating system structuring concept”, Comm. ACM, Vol. 17 (10), pp 549-557, 1974

Hoare78 Hoare, C.A.R., “Communicating sequential processes”, Comm. ACM, Vol. 21 (8), pp 666-677, 1978

Hodgson97 An interview with the Head of IT at I.T.L., Mr Richard Hodgson, 1997.

Hopper68 Hopper, G. M., Keynote Address at the inaugural History of Programming Languages conference, June 1-3, 1978, cited in [Wexelblat81].

Hopson96 Hopson, K.C., Ingram, S.E., Chan, P., “Developing Professional Java Applets”, Sams Publishing, 1996, ISBN: 1575210835

Horowitz83 Horowitz, E., “Fundamentals of Programming Languages”, Springer-Verlag, 1983.

Hudak89 Hudak, P., “Conception, Evolution and Application of Functional Programming Languages”, ACM Computing Surveys, Vol 21, pp 359-411, 1989.

IBM56 IBM Corporation, "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM", 956.

ICSE99 Proceedings of 21st International Conference on Software Engineering, “Preparing for the Software Century”, ACM PRES 1999, ISBN: 1-58113-074-0

IEEE80 IEEE Transactions on Communications, Special Issue on Computer Network Architectures and Protocols, Vol. 28 (4), April 1980.

ISO83 International Standards Organisation, “Basic Reference Model for Open Systems Interconnection”, ISO 7498, ISO, 1983

ISO90 International Organisation for Standardisation. Pascal. Technical report ISO 7185. ISO Geneva, 1991.

ISO92 International Standards Organisation, “Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use”, ISO/IEC JTC1/SC212/WG7 CD 10746-1, ISO, 1992

Iverson62 Iverson, K.E., “A Programming Language”, Wiley and Sons, 1962.

Jennings98 Jennings, N.R., Sycara, K.P. and Wooldridge, M., “A roadmap of agent research and development.”, in Autonomous Agents and Multi-Agent Systems, 1, pp 7-38, Kluwer Academic Publishers, 1999.

Johansen99 Johansen, D., interview in [Milojicic99], IEEE Concurrency, 1999.

Johansen99b Johansen, D., “Mobile Agent Applicability.”, In, Proceedings of the Mobile Agents 1998, Springer-Verlag LNCS series, Stuttgart, 9-11 September, 1998. Also in, Journal of Personal Technologies, Springer-Verlag, Vol 2, No. 2, 1999.

Jones83 Jones, M.B., Rashid, R.T., “Mach and Matchmaker: kernel and language support for object-oriented distributed systems”, ACM SIGPLAN Notices, Vol. 21 (11), pp 67-77

Jones97 Jones, M. Dr., Given in a presentation at the EPSRC Methodology Workshop held at Cambridge University, UK, 1997.

Jul88 Jul, E., Levy, H., Hutchinson, N., Black, A., “Fine-grained Mobility in the Emerald System”, ACM Transactions on Computer Systems, Vol 6 (2), 1988, pp 109-133.

Kernighan78 Kernighan, B.W., Ritchie, D.M., “The C Programming Language”, Prentice Hall, 1978.

Kiczales97 Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J-M., Irwin, J., “Aspect-Oriented Programming”, Proc. European Conf. on OOP (ECOOP), Springer-Verlag LNCS 1241, 1997

Knabe96 Knabe, F.C., “Language and compiler support for mobile agents”, PhD thesis, Carnegie Mellon University, 1996.

Knuth74 Knuth, D.E., “Structured programming with goto statements”, ACM Computing Surveys, 6 (4), pp 261-301, Dec. 1974

Kogure83 Kogure, M., Akao, Y., “Quality Function Deployment and CWQC in Japan”, Quality Progress, October 1983, pp 25-29.

Kotz99 Kotz, D., Gray, R.S., “Mobile code: The Future of the Internet”, in [Papaioannou/Minar99], 1999.

Kowalski79 Kowalski, R.A., “Logic for Problem Solving”, North-Holland, Amsterdam, 1979.

Kramer83 Kramer, J., Magee, J., Sloman, M., Lister, A., “CONIC: an integrated approach to distributed computer control systems”, IEE Proceedings, Part E, Vol 130 (1), pp 1-10, 1983.

Labrou94 Labrou, Y., Finin, T., “A Semantics Approach for KQML – a General Purpose Communication Language for Software Agents”, in proc. 3rd Int’l Conf. On Information and Knowledge Management (CIKM’94), 1994.

Lampson77 Lampson, B., Mitchell, J., Satterthwaite, E., "Report on the programming language Euclid", SIGPLAN Notices, 12 (2), Feb 1977

Lange98 Lange, D.B., Oshima, M., "Mobile Agents with Java: The Aglet API", World Wide Web Journal, 1998.

Lea93 Lea, R., Jacquement, C., Pillevesse, E., “COOL: System Support for Distributed Object-Oriented Programming”, Comm. ACM, Vol 36 (9), 1993, pp 37-46.

Leffler89 Leffler, S., McKusick, M., Karels, M. and Quartermain, J., “The Design and Implementation of the 4.3 BSD Unix Operating System”, Addison-Wesley, 1989

Lindholm99 Lindholm, T., Yellin, F., “The Java Virtual Machine Specification 2nd Edition”, Addison-Wesley, 1999. ; ISBN: 0201432943

Liskov81 Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Sheifler, R. and Snyder, A., “CLU Reference Manual”, Springer-Verlag, 1981.

Liskov83 Liskov, B., Sheifler, R., “Guardians and actions: linguistic support for robust distributed programs”, ACM TOPLAS, Vol 5 (3), pp 381-404, 1983

Liskov88 Liskov, B., “Distributed Programming in Argus”, Comm. ACM, Vol. 31 (3), pp 300-12, 1988

MacLennan87 MacLennan, B. J., "Principles of programming languages (Second Edition)", Holt, Rinehart & Winston, 1987.

MAL99 The Mobile Agents List, a repository of mobile agent systems, available at: http://www.informatik.uni-tuttgart.de/ipvr/vs/projekte/mole/mal/mal.html

Malamud91 Malamud, C., “Analysing DECnet / OSI Phase V”, Van Nostrand Rheinhold, NY, 1991

Martin99 Martin, D. L., Cheyer, A. J., and D. B. Moran, "The open agent architecture: A framework for building distributed software systems," Applied Artificial Intelligence, vol. 13, pp. 91--128, January-March, 1999.

McCall77 McCall, J.A., Richards, P.K.z, Walters, G.F., “Factors in Software Quality”, Rome Air Development Centre, RADC TR-77-369, 1977.

McCarthy60 McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine.", Comm ACM, 3 (4), pp 184-195, 1960.

McFayden76 McFayden, J.H., “Systems network architecture: An overview”, IBM Systems Journal, Vol 15 (1), pp 4-23, 1976.

McQuillan77 McQuillan, J., Walden, D., “The ARPA network design decision”, Computer Networks, 1 (3), pp 243-289, 1977

Mendelson64 Mendelson, E., “Introduction to Mathematical Logic”, Van Nostrand Reinhold, 1964.

Metcalfe76 Metcalfe, R.M., Boggs, D.R., “Ethernet: Distributed Packet Switching for local computer networks.”, Comm. ACM, 19 (7), pp 395-404, 1976

Milner90 Milner, R., Tofte, M., Harper, R.M., “The Definition of Standard ML”, MIT Press, Cambridge, MA, 1990.

Milojicic99 Milojicic, D., “Trend Wars: Mobile Agent Applications”, IEEE Concurrency, pp 80-90, July-September, 1999.

Minar98 Minar, N., “Designing an Ecology of Distributed Agents”, Masters Thesis, Media Lab, MIT, 1998.

Minar99 Minar, N., Gray, M., Roup, O., Krikorian, R., Maes, P., (1999), “Hive: Distributed Agents for Networking Things”, Proceedings of. ASA/MA ‘99.

Minar99b Private email correspondence with Nelson Minar, Hive team lead and chief architect, Dec 1999.

Mitchel79 Mitchel, J.G., Maybury, W., Sweet, R., “Mesa Language Reference Manual (V5.0)”, Tech. Report CSL-79-3, Xerox PARC, Palo Alto, CA.

Mobility98 Frequently Asked Questions (FAQ) of The Mobility List, 1998.Available at http://mobility.lboro.ac.uk/faq.html

Mobility99 The Mobility Mailing List – de facto mailing for discussion of mobility. Home page at: http://mobility.lboro.ac.uk

Molina98 Molina, A., Flores, M., Caballero, D., "Virtual Enterprises: A Mexican Case Study", published in [Camarinha-Matos98], 1998, pp159-170.

MSI99 MSI Research Institute, Final Report of the EPSRC Grant entitled “Manufacturing Software Interoperability: Steps towards Interoperating Distributed Objects”, EPSRC Grant No GR/M02446 (GR/K50504), Duration 01/05/95 – 30/04/99, 1999

Mullender93 Mullender, S.J. (ed), “Distributed Systems 2nd Edition”, ACM Press, 1993.

Naur63 Naur, P. et al (Eds.), "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6, pp. 1-17, 1963.

Naur78 Naur, P., “The European Side of the Last Phase of the Development of Algol 60”, SIGPLAN Notices 13 (8), pp 15-44, 1978.

Nelson91 Nelson, G., “Systems Programming with Modula-3”, Prentice-Hall, Englewood Cliffs, 1991.

OMG94 Object Management Group, “The Common Object Request Broker: Architecture and Specification”, OMG Inc., 492 Old Connecticut Path, Framingham, MA, USA, 1994

OMG99 Object Management Group, “IDL Syntax and Semantics”, OMG Inc., 492 Old Connecticut Path, Framingham, MA, USA, 1999, available at: http://www.omg.org/pub/orbrev/drafts/revised_99-08-01.idl

OSF92 OSF, “Introduction to OSF DCE”, Prentice-Hall, 1992

OSI84 Reference Model of Open Systems Interconnection for CCITT Applications, Malaga-Torremolinos, 1984

Osório98 Osório, A.L., Nuno, O., Camarinha-Matos, L., "Concurrent Engineering in Virtual Enterprises: the Extended CIM-FACE Architecture", published in [Camarinha-Matos98], pp171-184.

Ousterhout94 Ousterhout, J.K., “Tcl and the Tk toolkit”, Addison-Wesley, 1994.

Papaioannou/ Minar99 Papaioannou, T., Minar, N., “Mobile Agents in the Context of Competition and Cooperation”, Proc. of MAC3 workshop, part of Autonomous Agents ‘99 conference, Seattle, 1999.

Papaioannou98 Papaioannou, T., Edwards, J.M., “Mobile Agent Technology Enabling the Virtual Enterprise: A Pattern for Database Query”, in notes of Agent Based Manufacturing Workshop, part of the International Technical Conference Autonomous Agents '98.

Papaioannou99 Papaioannou, T., Edwards, J.M., “Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems”, International Journal of Robotics and Autonomous Systems, 27, pp 45-57, 1999.

Papaioannou99b Papaioannou, T., Edwards, J.M., “Mobile Agent Technology in Support of Sales Order Processing in the Virtual Enterprise”, in [Camarinha-Matos et al]

Papaioannou99c Papaioannou, T., “Mobile Agents: Are They Useful for Establishing a Virtual Presence in Space?”, in notes of Adjustable Autonomy Symposium, part of the AAAI Spring Symposium Series, Stanford University, 1999.

Papaioannou2000 Papaioannou, T., Edwards, J.M., “Manufacturing Systems Integration and Agility: Can Mobile Agents Help?”, accepted for publication in Journal of Integrated Computers-Aided Engineering, IOS Press. To appear in 2000.

Papaioannou2000b Papaioannou, T., Edwards, J.M., “Towards Understanding and Evaluating Mobile Code Systems”, accepted for publication in Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers. To appear in 2000.

Papastavrou99 Papastavrou, S., Samaras, G., Pitoura, E., “Mobile Agents for WWW Distributed Database Access”, in Proc. IEEE International
Conference on Data Engineering (ICDE99), 1999.

Parnas72a Parnas, D.L., “A technique for software module specification with examples”, Comm. ACM, Vol 15 (5), pp 330-336, 1972.

Parnas72b Parnas, D.L., “On the criteria to be used in decomposing systems into modules”, Comm. ACM, Vol 15 (12), pp 1053-1058, 1972.

Peine98 Peine, H., Stolpmann, T., “The Architecture of the Ara Platform for Mobile Agents”, in [Rothermel97], pp 50-61.

Perlis58 Perlis, A., Samelson, K., "Preliminary Report - International Algebraic Language", Comm. ACM 1 (12) pp. 8-22, 1958.

Peters82 Peters, T., Waterman, R.H., Jr, “In search of Excellence”, HarperCollins, 1982.

Peters85 Peters, T., Austin, N., “A Passion for Excellence – the Leadership difference”, HarperCollins, 1985.

Picco98 Picco, G.P., “Understanding, Evaluating, Formalizing, and Exploiting Code Mobility”, PhD thesis, Politecnico di Torino, 1998

Picco98b Picco, G.P., Baldi, M., “Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications”, In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto (Japan), R. Kemmerer and K. Futatsugi, eds., April 1998, IEEE CS Press, ISBN 0-8186-8368-6, pp. 146-155, 1998.

Pinker95 Pinker, S., “The Language Instinct”, Harper Collins, 1995, ISBN: 0060976519.

Pouzin73 Pouzin, L., “Presentation and major design aspects of the CYCLADES computer network”, in Proc. 3rd ACM-IEEE Communications Symposium, pp 80-87, 1973

Pratt84 Pratt, T.W., "Programming Languages, Design and Implementation, 2nd edition", Prentice-Hall, 1984.

Raj91 Raj, R.K., Tempero, E., Levy, H.M., Black, A.P., Hutchinson, N.C., and Jul, E., “Emerald: A general purpose programming language”, Software-Practice and Experience, Vol 21 (1), 1991.

Rashid81 Rashid, R., Robertson, G., “Accent: a communications oriented network operating system kernel”, ACM Operating Systems Review, Vol 15 (5), pp 64-75, 1981

Rashid86 Rashid, R., “From RIG to Accent to Mach: the evolution of a network operating system”, Proc. ACM/IEEE Computer Society Fall Joint Conference, ACM, 1986.

Raymond98 Raymond, E.S., “The Cathedral and the Bazaar”, Version 1.40, 1998/08/11, http://www.tuxedo.org/~esr/

Redmond97 Redmond, F., III, "Dcom: Microsoft Distributed Component Object Model", IDG Books Worldwide, ISBN: 0764580442

Reed79 Reed, D.P., Kanodia, R.K., "Sychronisation with Eventcounts and Sequences", Comm. ACM, Vol. 22 (2), pp 3-23, 1979.

Ritchie74 Ritchie, D.M., Thompson, K., "The UNIX time-sharing system", Comm. ACM, Vol 17 (7), pp 365-375, 1974

Roberts70 Roberts, L.G., Wessler, B.T., "Computer network development to achieve resource sharing", in Proc. SJCC, pp 543-549, 1970.

Rose90 Rose, M.T., "The Open Book: a practical perspective on OSI", Prentice-Hall, 1990

Rothermel97 Rothermel, K., Popescu-Zeletin, R., Eds, "Mobile Agents: 1st International Workshop MA'97", Lecture Notes in Computer Science, Vol 1219, Springer-Verlag, 1997.

Rothermel98 Rothermel, K., Hohl, F., Eds, "Mobile Agents, 2nd Int'l Workshop MA '98", Lecture Notes in Computer Science, Vol 1477, Springer-Verlag, 1998.

Rus97 Rus, D., Gray, R., Kotz, D., "*Transportable information agents*", Journal of Intelligent Information Systems, 9:215-238, 1997

Scanlon97 Scanlon, E., "Suggestions for Case Study Research Methods", http://www.gwbssw.wustl.edu/~csd/evaluation/casestudy/caseguide.html

Shock80 Shock, J.F., "An annotated Bibliography on Local Computer Networks", XEROX Palo Alto Research Center, 1980.

Shrivastava-et-al Shrivastava, S.K., Dixon, G., Parrington, G.D., Hedayati, F., Wheater, S., Little, M., "The Design and Implementation of Arjuna", Proc. 3rd Conference on Object Oriented Programming, Nottingham.

Shroeder93 Shroeder, M, D., "A State-of-the-Art Distributed System: Computing with BOB.", In Distributed Systems, 2nd ed., S. Mullender, ed., ACM Press, 1993

Simon96 Simon, E., "Distributed Information Systems – from client/server to distributed multimedia", McGraw-Hill, 1996.

Sloman85 Sloman, M., Kramer, J., Magee, J., "The Conic toolkit for building distributed systems", Proc. 6th IFAC Workshop on Distributed Computer Control Systems, California, Pergamon Press, 1985

Sloman87 Sloman, M., Kramer, J., "Distributed Systems and Computer Networks", Prentice-Hall, 1987

Solingen99 Solingen, R.V., Berghout, E., (1999), "The Goal/Question/Metric Method",McGraw Hill, ISBN 0-07-709553-7

SSA95 System Software Associates Inc., "BPCS Client/Server Distributed Object Computing Architecture", Technical Report, 1995.

Stallings87 Stallings, W., "Handbook of Computer Communications Standards", Vol 1, Macmillan, NY, 1987

Stamos86 Stamos, J.W., "Remote Evaluation", Technical Report TR-354, MIT, 1986

Straßer96 Straßer, M, Baumann, J., Hohl, F., (1996), "Mole - A java Based Mobile Agent System", in Proc. ECOOP'96 workshop on Mobile Object Systems.

Strauss90 Strauss, A. & Corbin, J. (1990) "Basics of Qualitative Research", Newbury Park, CA.: Sage Publications.

Stroustrup92 Stroustrup, B., "The C++ Programming Language", 2nd edition, Addison-Wesley, Reading, MA, 1992.

Sun89 Sun Microsystems Inc., "NFS: Network File System Protocol Specification", Tech. Report RFC 1094, file available for anonymous ftp from ftp://nic.ddn.mil, directory /usr/pub/RFC, 1989

Sun97 Sun Microsystems Inc., "JavaBeans Component Framework Specification", Revision 1.01, JDK 1.1, July 1997.

Sun98 Sun Microsystems Inc., "Java Remote Method Invocation Specification", Revision 1.50, JDK 1.2, October 1998.

Sun98b Sun Microsystems Inc., "Object Serialisation Specification", JDK 1.1, October 1998, available at http://java.sun.com/products /jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html

Sun99 Sun Microsystems Inc., "Enterprise Javabeans Specification", Version 1.1, 1999, available at http://java.sun.com/products/ejb/docs.html

Tanenbaum96 Tanenbaum, A.S., "Computer Networks – 3rd Edition", Prentice-Hall, 1996.

Teitelman84 Teitelman, W., "A tour through Cedar", IEEE Software, Vol. 1 (2), pp 44-73, 1984

Thiel91 Thiel, G., "Locus Operating System, a transparent system", Computer Communications, Vol 14 (6), 1991, pp336-346.

Thompson96 Thompson, S., “Haskell: The Craft of Functional Programming”, Addison-Wesley, 1996.

Tsichritzis85 Tsichritzis, D., “*Objectworld*”, Office Automation, Springer-Verlag, 1985

Turing36 Turing, A.M., “On Computable Numbers, with an application to the Entscheidungsproblem”, Proc. London Mathematical Society, Vol 2 (42), pp 230-265, 1936.

Turner85 Turner, D.A., “Miranda: A non-strict functional language with polymorphic types”, in Functional Programming Languages and Computer Architercture, Lecture Notes in Computer Science 201, pp 1-16, Springer-Verlag, 1985.

UCI96 “MESSENGERS: A Distributed Computing Environment for AutonomousObjects” UCI Technical Report: TR-96-20, 1996, available from http://www.ics.uci.edu/~bic/messengers/

Vigna98 Vigna, G., ed, “Mobile Agents and Security”, LNCS Vol 1419, Springer-Verlag, 1998.

Vinoski99 Vinoski, S., Chief Architect at Iona Technologies Inc, comment made in dist-obj mailing list. Thu, 15 Jul, 1999.

Waldo94 Waldo, J., Wyant, G., Wollrath, A., Kendall, S., “A note on distributed computing”, Sun Microsystems Technical Report SML 94-29, 1994.

Walsh85 Walsh, D., Lyon, B., Sager, G., Change, J.M., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R. and Weiss, P., “Overview of the Sun Network File System”, Proc. of the Winter Usenix Conference, 1985.

Watt96 Watt, S., “Pride and prejudice: four decades of LISP”, in [Woodman96], pp 235-254, 1996

Wecker80 Wecker, S., “DNA: the digital network architecture”, in [IEEE80]

Weiser91 Weiser, M., “The Computer for the 21st Century”, Scientific American, Vol 265 (3), pp 94-104, 1991.

Wexelblat81 Wexelblat, R.L., ed, "History of Programming Languages", ACM Monograph Series, Academic Press, 1981.

White94 White, J.E, “Telescript technology: the foundation for the electronic marketplace”, White Paper, General Magic, Inc. USA, 1994.

White96 White, J., “Telescript Technology: Mobile Agents”, In Software Agents, Bradshaw, J., Ed., AAAI Press/MIT Press, 1996.

Whitmire97 Whitmire, S.A., “Object-oriented design measurement”, Wiley and Sons, 1997, ISBN:0-471-13417-1

Wilson93 Wilson, L.B., Clark, R.G., “Comparative Programming Languages”, Addison-Wesley, 1993.

Wirth77 Wirth, N., “Modula: A language for modular programming”, software Practice Experience, 7 (1), Jan 1977.

Wirth82 Wirth, N., "Type Extensions", ACM Transactions on Programming Languages and Systems, 10 (2), pp. 204-214, February 1988.

Wong97 Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., Peet, B., “Concordia: An infrastructure for collaborating mobile agents”, in Proc. First Int’l Workshop on Mobile Agents ’97, Springer-Verlag, 1997

Woodman96 Woodman, M., “Programming Language Choice, Practice and Experience”, Thomson Computer Press, 1996.

Wooldridge99 Wooldridge, M., Jennings, N.R., Kinny, D., “A Methodology for Agent-Oriented Analysis sand Design”, in Proc. 3rd Annual Conf. On Autonomous Agents, Eds, Etzioni, O., Müller, J.P., Bradshaw, J.M., ACM Press, 1999.

Wright96 Wright, D.T., Burns, N.D., "Impact of Globalisation on organisational Structure and Performance", Proc. of the Organisational Management Division, International Association of Management 14th Annual Conference. Toronto, Canada, August 2-6, 1996, pp. 58-63

Yin94 Yin, R.K., “Case Study Research”, Sage Publications, 1994

Yourdon79 Yourdon, E., Constantine, L.L., “Structured Design”, Prentice-Hall 1979.

Zak98 Zak, D., “Programming With Microsoft Visual Basic 6.0”, Microsoft Press, 1998.

Zimmerman80 Zimmerman, H., “OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection, in [IEEE80].

Appendices

Appendix A

Program listing of an example OrderAgent:

```
package uk.ac.lboro.todd.aglets.mascenario;

import uk.ac.lboro.todd.aglets.mascenario.tasks.*;
import uk.ac.lboro.todd.aglets.*;
import uk.ac.lboro.todd.aglets.order.*;
import uk.ac.lboro.todd.aglets.utils.*;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;
import java.util.Date;

/**
 * A simple QueryAglet that can be created by a Master and tasked
 * with tracking down the stock levels of a product from a list of
 * hosts.
 *
 * @version   2.1   10/11/98    Changed from a properties lookup for
 *                              the DataSource to multicast messaging
 *  version   2.0   21/10/98    Most of the required logic has now
 *                              been refactored and shifted to the
 *                              Task classes. Allows for far more
 *                              modularity.
 *  version   1.2   18/10/98    Query can now handle missing data
 *                              sources and also the addition of
 *                              subsequent tasks, after the
 *                              completion of the first one.
 *  version   1.1   08/10/98    Query aglet is now able to complete
 *                              Itinerary and request a retraction
 *                              Removed MakeRequest and added it to
 *                              StockRequestTask. Makes more sense.
 *  version   1.01  25/09/98    Added capability to create with
 *                              details and receive an Itinerary.
 *  version     1.00        23/09/98    First attempt.
 *
 * @author      Todd Papaioannou
 */

public class QueryAglet extends BlindAglet {

    // Our data variables
    AgletProxy dataProxy = null;
    ResultSet resSet = null;
    AgletProxy mProxy = null;
    SlaveItin itin = null;
    Order order = null;

    // Do some tasks when the aglet is created
    public void onCreation(Object init) {

        // Pass up the hierarchy
```

```
        super.onCreation(init);

        SlaveDetails det = (SlaveDetails)init;

        // Must make a note of the master here
        mProxy = det.getMaster();

        // Initialise our important internals
        resSet = new ResultSet(getAgletID());
        order = det.getOrder();

        // Add our own listener and adapter
        addMobilityListener(
            new MobilityAdapter() {

                int counter = 0;

                // Using this as a safety check in case we get caught
                // in a loop in the same host
                public void onArrival(MobilityEvent event) {

                    if (counter > 1)
                        System.out.println("ACounter = " +
                              new Integer(counter).toString());
                    counter++;

                    if (counter > 3) {
                        System.out.println("Self destructing!");
                        try {
                            event.getAgletProxy().dispose();
                        } catch (Exception e) {
                            System.out.println(e.toString());
                        }
                    }
                }

                public void onDispatching(MobilityEvent event) {
                    counter = 0;
                }

                public void onReverting(MobilityEvent event) {
                    appendMessage("Being retracted by Master to
                                                    homebase.");
                }
            }

        ); /* End of Adapter */

    }

    // Test run
    public void run() {

        //System.out.println("\nInto run");

        // Just a safety check, in case of delay
        while (itin == null) {
            for (int i = 0; i < 3; i++) {

                waitMessage(1 * 1000);
            }
```

```
        }

        // Do we have an itinerary and is this the last stop?
        if ((itin != null) && itin.atLastDestination()) {

            // Let's get a reference to the final Task object.
            GenericTask task =
                        (GenericTask)itin.getTaskAt(itin.size()-1);

            try {
                task.finishTasks(itin);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    /**
     * Handle ourselves being killed gracefully
     *
    public void onDisposing() {

        // Clear up and get rid of our itinerary
        itin.clear();
        removeMobilityListener(itin);
    }*/

    /**
     * Returns true if the current host is our origin
     */
    public boolean atHome() {

        if (getAgletInfo().getOrigin().equals(getAgletContext(). \
            getHostingURL().toString()))
            return true;
        else
            return false;
    }

    /**
     * Allows a slave to contact it's master and ask for a
     * retraction. Useful since the Master has no idea where the
     * Slave might have ended up.
     */
    public void returnHome() {

        try {
            Message msg = new Message("RetractMe");
            msg.setArg("url", getAgletContext().getHostingURL());
            msg.setArg("id", getAgletID());
            mProxy.sendOnewayMessage(msg);
        } catch (InvalidAgletException iae) {
            System.out.println("1 " + iae.toString());
        } catch (Exception e) {
            System.out.println("2 " + e.toString());
        }
     }

    /**
     * Find out who is the data source in this context
     */
```

```
    public boolean whoSource() {

        try {
            ReplySet set = getAgletContext().multicastMessage
                                        (new Message("DataSource?"));

            // Give any sluggards a chance
            while (!set.isAnyAvailable())
                waitMessage(1*10);

            FutureReply future = set.getNextFutureReply();
            Object reply = future.getReply();
            AgletID aid = (AgletID)reply;
            dataProxy = getAgletContext().getAgletProxy(aid);

        } catch (NotHandledException ex) {
            System.out.println(ex);
            dataProxy = null;
        } catch (MessageException ex) {
            System.out.println(ex);
            dataProxy = null;
        }

        if (dataProxy != null)
            return true;
        else
            return false;
    }

    /**
     * Attempt to handle any incoming messages
     */
    public boolean handleMessage(Message msg) {

        if (msg.sameKind("Itinerary")) {
            itin = (SlaveItin)msg.getArg();
            appendMessage("Itinerary received, starting trip.");
            itin.startTrip();
        } else {
            System.out.println(msg.toString());
            return false;
        }

        return true;
    }

    /**
     * Override super class method to allow for easy redirection
     * during testing.
     */
    public void appendMessage(String text) {
        System.out.println("[" + getName() + "] " + text);
    }

    /**
     * Return the current order we are dealing with
     */
    public Order getOrder() {
        return order;
    }
```

```
    /**
     * Return our current result set
     */
    public ResultSet getResults() {
        return resSet;
    }

    /**
     * Allow someone to try to clear our result set
     */
    public void clearResults() {
        resSet = null;
    }

    /**
     * Return a reference to our Master's proxy
     */
    public AgletProxy getMasterProxy() {
        return mProxy;
    }

    /**
     * Return a reference to the DataAglet's proxy
     */
    public AgletProxy getDataProxy() {
        return dataProxy;
    }

    /**
     * Return a reference to our Itinerary
     */
    public SlaveItin getItin() {
        return itin;
    }

} /* End of Class */
```

Appendix B

Program listing of an example Agent Task:

```
package uk.ac.lboro.todd.aglets.mascenario.tasks;

import uk.ac.lboro.todd.aglets.*;
import uk.ac.lboro.todd.aglets.utils.*;
import uk.ac.lboro.todd.aglets.mascenario.*;
import uk.ac.lboro.todd.aglets.order.*;

import com.ibm.aglet.*;
import com.ibm.agletx.util.*;
import java.net.URL;

/**
 * StockRequestTask - a task that allows an agent to make a request
 * to a DataSource aglet. The request is encapsulated within the
 * Order object the slave carries around with it.
 *
 * @version   2.1   04/11/98    First attempt with MA instead
 *                              of MO's. Added evalResult().
 * @version   2.0   21/10/98    Massive refactoring of the
 *                              code. Very little of the behaviour of
 *                              the Aglet relies on code in run()
 *                              The addition of finishTasks allows
 *                              for a much simpler and more modular
 *                              approach to design of Slave agents.
 * @version   1.11  08/10/98    MakeRequest has been added from
 *                              QueryAglet. Makes more sense.
 * @version   1.10  08/10/98    StockRequest now fully functional
 * @version   1.00  28/09/98    First attempt.
 *
 * @author      Todd Papaioannou
 */

public class StockRequestTask extends GenericTask {

    /**
     * Use this to allow us a better view of what goes on at a host
     */
    static boolean pause = true;

    // Our owner aglet
    QueryAglet qag = null;
    Result result = null;

    /**
     * The actual work associated with this Task.
     */
    public void execute(SeqItinerary itin) throws Exception {

        // Find out who the data source is
        AgletProxy proxy = itin.getOwnerAglet();
        qag = (QueryAglet)proxy.getAglet();
        URL currentHost = qag.getAgletContext().getHostingURL();

        // Is this the last desination?
        if (itin.atLastDestination() == false) {
```

```
            // We must still have some tasks to do.
            // Is there a data source handy?
            if (qag.whoSource() != true) {
                qag.appendMessage("No damn data source!");

                // Are we actually at the last address?
                if (!currentHost.toString().
                          equals(itin.getAddressAt(itin.size()-1)) ) {

                    // Make it easier to see what's actually going on
                    if (pause)
                        qag.waitMessage(2*1000);
                    qag.appendMessage("Proceeding to next stop on \
                                                          Itinerary");
                }
            } else {
                qag.appendMessage("Found a data source.");

                // Get our info from the data source
                makeRequest();
                qag.appendMessage("Finished Request, evaluating \
                                                            results.");
                evalResult();
            }
        }
    } // End of execute

    /**
     * Make a request for an Order to be checked.
     */
    public void makeRequest() {

        try {

            Object reply = qag.getDataProxy().sendMessage(
                  new Message("Order", new NamedOrder(qag.getName(),
                                                qag.getOrder())));
            result = (Result)reply;

        } catch (InvalidAgletException ex) {
            System.out.println(ex);
        } catch (NotHandledException ex) {
            System.out.println(ex);
        } catch (MessageException ex) {
            System.out.println("[ERROR] Make Request Failed because \
                                       of:\n" + ex.getException());
            System.out.println(ex);
        }

        // Let's put some artificial pausing in. Looks good for the
        // humans!
        if (pause) {
            for (int i=0; i < 160; i++) {
               System.out.print(".");
            }
            System.out.println("\n");
        }

    } // End of makeRequest
```

```
    /**
     * Can this host satisfy our order?
     */
    private void evalResult() {

        boolean success = false;

        if (result.getIndicator() == Result.YES) {

            qag.appendMessage("We have a RESULT!");
            qag.appendMessage("Result " + result.getHost() + " will \
                                              satisfy this order.");
            success = true;
        }

        // Add this result to our set for future reference.
        qag.getResults().addResult((Result)result);

        if (success) {
            commitOrder();
        } else {
            qag.appendMessage("Current host cannot satisfy order. \
                                              Going to next host.");
        }
    }

    // This routine allows us to attempt to commit and order
    private void commitOrder() {

        try {

            String reply = (String)qag.getDataProxy().sendMessage(
                  new Message("Commit", new NamedOrder(qag.getName(),
                                                  qag.getOrder())));

            // We have successfully committed the Order
            if (reply.equals("Committed")) {

                qag.appendMessage("Order successfully committed.");
                qag.getMasterProxy().sendOnewayMessage(new Message
                     ("Committed", qag.getOrder().getOrderNumber()));
                qag.appendMessage("Tasks have been completed. \
                                             Disposing of myself.");

                // Kill ourselves
                qag.dispose();

            } else if (reply.equals("OutOfStock")) {
                qag.appendMessage("Out of Stock!");
                qag.getMasterProxy().sendOnewayMessage(new Message
                    ("OutOfStock", qag.getOrder().getOrderNumber()));
                qag.dispose();
            } else {
                qag.appendMessage("Something messed up! Getting rid \
                                                      of myself.");
                qag.dispose();
            }

        } catch (InvalidAgletException ex) {
            System.out.println(ex);
        } catch (NotHandledException ex) {
```

```
            System.out.println(ex);
        } catch (MessageException ex) {
            System.out.println("[ERROR] Make Request Failed because \
                                    of:\n" + ex.getException());
            System.out.println(ex);
        }

    }

    // Must define this since it's abstract
    public void finishTasks(SeqItinerary itin) throws Exception {
    }

} /* End of Class */
```

Background Readings on Agents

Introduction to Agents [Adobe Acrobat pdf file]

Java, Agents, and Heteroeneous Information Sources

The Same in Postscript (.ps)

Agent Communication Languages [Adobe Acrobat pdf]

MAC3 Mobile Agent Workshop 1999 [Adobe Acrobat pdf]

KQML Architecture (and CORBA) [postscript ps]

Software Agents

Chapter 1

An Introduction to Software Agents

Jeffrey M. Bradshaw

Since the beginning of recorded history, people have been fascinated with the idea of non-human agencies.[1] Popular notions about androids, humanoids, robots, cyborgs, and science fiction creatures permeate our culture, forming the unconscious backdrop against which software agents are perceived. The word "robot," derived from the Czech word for drudgery, became popular following Karel Capek's 1921 play *RUR: Rossum Universal Robots.* While Capek's robots were factory workers, the public has also at times embraced the romantic dream of robots as "digital butlers" who, like the mechanical maid in the animated feature "The Jetsons," would someday putter about the living room performing mundane household tasks. Despite such innocuous beginnings, the dominant public image of artificially intelligent embodied creatures often has been more a nightmare than a dream. Would the awesome power of robots reverse the master-slave relationship with humans? Everyday experiences of computer users with the mysteries of ordinary software, riddled with annoying bugs, incomprehensible features, and dangerous viruses reinforce the fear that the software powering autonomous creatures would pose even more problems. The more intelligent the robot, the more capable of pursuing its own self-interest rather than its master's. The more humanlike the robot, the more likely to exhibit human frailties and eccentricities. Such latent concerns cannot be ignored in the design of software agents—indeed, there is more than a grain of truth in each of them!

Though automata of various sorts have existed for centuries, it is only with the development of computers and control theory since World War II that anything resembling autonomous agents has begun to appear. Norman (1997) observes that perhaps "the most relevant predecessors to today's intelligent agents are servomechanisms and other control devices, including factory control and the automated takeoff, landing, and flight control of aircraft." However, the agents now being contemplated differ in important ways from earlier concepts.

Significantly, for the moment, the momentum seems to have shifted from hardware to software, from the atoms that comprise a mechanical robot to the bits that make up a digital agent (Negroponte 1997).[2]

Alan Kay, a longtime proponent of agent technology, provides a thumbnail sketch tracing the more recent roots of software agents:

> "The idea of an agent originated with John McCarthy in the mid-1950's, and the term was coined by Oliver G. Selfridge a few years later, when they were both at the Massachusetts Institute of Technology. They had in view a system that, when given a goal, could carry out the details of the appropriate computer operations and could ask for and receive advice, offered in human terms, when it was stuck. An agent would be a 'soft robot' living and doing its business within the computer's world." (Kay 1984).

Nwana (1996) splits agent research into two main strands: the first beginning about 1977, and the second around 1990. Strand 1, whose roots are mainly in distributed artificial intelligence (DAI), "has concentrated mainly on deliberative-type agents with symbolic internal models." Such work has contributed to an understanding of "*macro* issues such as the interaction and communication between agents, the decomposition and distribution of tasks, coordination and cooperation, conflict resolution via negotiation, etc." Strand 2, in contrast, is a recent, rapidly growing movement to study a much broader range of agent types, from the moronic to the moderately smart. The emphasis has subtly shifted from *deliberation* to *doing*; from *reasoning* to *remote action.* The very diversity of applications and approaches is a key sign that software agents are becoming mainstream.

The gauntlet thrown down by early researchers has been variously taken up by new ones in distributed artificial intelligence, robotics, artificial life, distributed object computing, human-computer interaction, intelligent and adaptive interfaces, intelligent search and filtering, information retrieval, knowledge acquisition, end-user programming, programming-by-demonstration, and a growing list of other fields. As "agents" of many varieties have proliferated, there has been an explosion in the use of the term without a corresponding consensus on what it means. Some programs are called agents simply because they can be scheduled in advance to perform tasks on a remote machine (not unlike batch jobs on a mainframe); some because they accomplish low-level computing tasks while being instructed in a higher-level of programming language or script (Apple Computer 1993); some because they abstract out or encapsulate the details of differences between information sources or computing services (Knoblock and Ambite 1997); some because they implement a primitive or aggregate "cognitive function" (Minsky 1986, Minsky and Riecken 1994); some because they manifest characteristics of distributed intelligence (Moulin and Chaib-draa 1996); some because they serve a mediating role among people and programs (Coutaz 1990; Wiederhold 1989; Wiederhold 1992); some because they perform the role of an "intelligent assistant" (Boy 1991, Maes 1997) some because they can migrate in a self-directed way from computer to computer

(White 1996); some because they present themselves to users as believable characters (Ball et al. 1996, Bates 1994, Hayes-Roth, Brownston, and Gent 1995); some because they speak an agent communication language (Genesereth 1997, Finin et al. 1997) and some because they are viewed by users as manifesting intentionality and other aspects of "mental state" (Shoham 1997).

Out of this confusion, two distinct but related approaches to the definition of agent have been attempted: one based on the notion of agenthood as an *ascription* made by some person, the other based on a *description* of the attributes that software agents are designed to possess. These complementary perspectives are summarized in the section "What Is a Software Agent." The subsequent section discusses the "why" of software agents as they relate to two practical concerns: 1) simplifying the complexities of distributed computing and 2) overcoming the limitations of current user interface approaches. The final section provides a chapter by chapter overview of the remainder of the book.

What Is a Software Agent?

This section summarizes the two definitions of an agent that have been attempted: agent as an ascription, and agent as a description.

'Agent' as an Ascription

As previously noted, one of the most striking things about recent research and development in software agents is how little commonality there is between different approaches. Yet there is something that we intuitively recognize as a "family resemblance" among them. Since this resemblance cannot have to do with similarity in the details of implementation, architecture, or theory, it must be to a great degree a function of the eye of the beholder.[3] "Agent is that agent does"[4] is a slogan that captures, albeit simplistically, the essence of the insight that agency cannot ultimately be characterized by listing a collection of *attributes* but rather consists fundamentally as an *attribution* on the part of some person (Van de Velde 1995).[5]

This insight helps us understand why coming up with a once-and-for-all definition of agenthood is so difficult: one person's "intelligent agent" is another person's "smart object"; and today's "smart object" is tomorrow's "dumb program." The key distinction is in our expectations and our point of view. The claim of many agent proponents is that just as some algorithms can be more easily expressed and understood in an object-oriented representation than in a procedural one (Kaehler and Patterson 1986), so it sometimes may be easier for developers and users to interpret the behavior of their programs in terms of agents rather than as more run-of-the-mill sorts of objects (Dennett 1987).[6]

The *American Heritage Dictionary* defines an agent as "one that acts or has

the power or authority to act… or represent another" or the "means by which something is done or caused; instrument." The term derives from the present participle of the Latin verb *agere:* to drive, lead, act, or do.

As in the everyday sense, we expect a software agent to act on behalf of someone to carry out a particular task which has been delegated to it.[7] But since it is tedious to have to spell out every detail, we would like our agents to be able to infer what we mean from what we tell it. Agents can only do this if they "know" something about the context of the request. The best agents, then, would not only need to exercise a particular form of expertise, but also take into account the peculiarities of the user and situation.[8] In this sense an agent fills the role of what Negroponte calls a "digital sister-in-law:"

> "When I want to go out to the movies, rather than read reviews, I ask my sister-in-law. We all have an equivalent who is both an expert on movies and an expert on us. What we need to build is a digital sister-in-law.
>
> In fact, the concept of "agent" embodied in humans helping humans is often one where expertise is indeed mixed with knowledge of you. A good travel agent blends knowledge about hotels and restaurants with knowledge about you… A real estate agent builds a model of you from a succession of houses that fit your taste with varying degrees of success. Now imagine a telephone-answering agent, a news agent, or an electronic-mail-managing agent. What they all have in common is the ability to model you." (Negroponte 1997).

While the above description would at least seem to rule out someone claiming that a typical payroll system could be regarded as an agent, there is still plenty of room for disagreement (Franklin and Graesser 1996). Recently, for example, a surprising number of developers have re-christened existing components of their software as agents, despite the fact that there is very little that seems "agent-like" about them. As Foner (1993) observes:

> "… I find little justification for most of the commercial offerings that call themselves agents. Most of them tend to excessively anthropomorphize the software, and then conclude that it must be an agent because of that very anthropomorphization, while simultaneously failing to provide any sort of discourse or "social contract" between the user and the agent. Most are barely autonomous, unless a regularly-scheduled batch job counts. Many do not degrade gracefully, and therefore do not inspire enough trust to justify more than trivial delegation and its concomitant risks."[9]

Shoham provides a practical example illustrating the point that although anything *could* be described as an agent, it is not always advantageous to do so:

> "It is perfectly coherent to treat a light switch as a (very cooperative) agent with the capability of transmitting current at will, who invariably transmits current when it believes that we want it transmitted and not otherwise; flicking the switch is simply our way of communicating our desires. However, while this is a coherent view, it does not buy us anything, since we essentially understand the mechanism sufficiently to have a simpler, mechanistic description of its behavior." (Shoham 1993).[10]

Physical Stance	Predict based on physical characteristics and laws
Design Stance	Predic t based on what it is designed to do
Intentional Stance	Precit based on assumption of rational agency

Table 1. Dennett's three predictive stances (from Sharp 1992, 1993).

Dennett (1987) describes three predictive stances that people can take toward systems (table 1). People will choose whatever gives the most simple, yet reliable explanation of behavior. For natural systems (e.g., collisions of billiard balls), it is practical for people to predict behavior according to physical characteristics and laws. If we understand enough about a *designed* system (e.g., an automobile), we can conveniently predict its behavior based on its functions, i.e., what it is designed to do. However as John McCarthy observed in his work on "advice-takers" in the mid-1950's, "at some point the complexity of the system becomes such that the best you can do is give advice" (Ryan 1991). For example, to predict the behavior of people, animals, robots, or agents, it may be more appropriate to take a stance based on the assumption of rational agency than one based on our limited understanding of their underlying blueprints.[11]

Singh (1994) lists several pragmatic and technical reasons for the appeal of viewing agents as intentional systems:

> "They (i) are natural to us, as designers and analyzers; (ii) provide succinct descriptions of, and help understand and explain, the behaviour of complex systems; (iii) make available certain regularities and patterns of action that are independent of the exact physical implementation of the agent in the system; and (iv) may be used by the agents themselves in reasoning about each other."

'Agent' As a Description

A more specific definition of "software agent" that many agent researchers might find acceptable is: a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes (Shoham 1997). The requirement for continuity and autonomy derives from our desire that an agent be able to carry out activities in a flexible and intelligent manner that is responsive to changes in the environment without requiring constant human guidance or intervention. Ideally, an agent that functions continuously in an environment over a long period of time would be able to learn from its experience. In addition, we expect an agent that inhabits an environment with other agents and processes to be able to communicate and cooperate with them, and perhaps move from place to place in doing so.

All this being said, most software agents today are fairly fragile and special-purpose beasts, no one of which can do very much of what is outlined above in a generic fashion. Hence the term "software agent" might best be viewed as an umbrella term that covers a range of other more specific and limited agent types (Nwana 1996). Though as individuals the capabilities of the agents may be rather restricted, in their aggregate they attempt to simulate the functions of a primitive "digital sister-in-law," as particular ones intimately familiar with the user and situation exchange knowledge with others who handle the details of how to obtain needed information and services. Consistent with the requirements of a particular problem, each agent might possess to a greater or lesser degree attributes like the ones enumerated in Etzioni and Weld (1995) and Franklin and Graesser (1996):

- *Reactivity*: the ability to selectively sense and act
- *Autonomy*: goal-directedness, proactive and self-starting behavior
- *Collaborative behavior*: can work in concert with other agents to achieve a common goal
- *"Knowledge-level" (Newell 1982) communication ability:* the ability to communicate with persons and other agents with language more resembling human-like "speech acts" than typical symbol-level program-to-program protocols
- *Inferential capability*: can act on abstract task specification using prior knowledge of general goals and preferred methods to achieve flexibility; goes beyond the information given, and may have explicit models of self, user, situation, and/or other agents.
- *Temporal continuity:* persistence of identity and state over long periods of time[12]
- *Personality*: the capability of manifesting the attributes of a "believable" character such as emotion
- *Adaptivity*: being able to learn and improve with experience
- *Mobility*: being able to migrate in a self-directed way from one host platform to another.

To provide a simpler way of characterizing the space of agent types than would result if one tried to describe every combination of possible attributes, several in the agent research community have proposed various classification schemes and taxonomies.

For instance, AI researchers often distinguish between *weak* and *strong* notions of agency: agents of the latter variety are designed to possess explicit mentalistic or emotional qualities (Shoham 1997; Wooldridge and Jennings 1995). From the DAI community, Moulin and Chaib-draa have characterized agents by degree of problem-solving capability:

> "A *reactive* agent reacts to changes in its environment or to messages from other agents.… An *intentional* agent is able to reason on its intentions and beliefs, to create plans of actions, and to execute those plans.… In addition to intentional agent

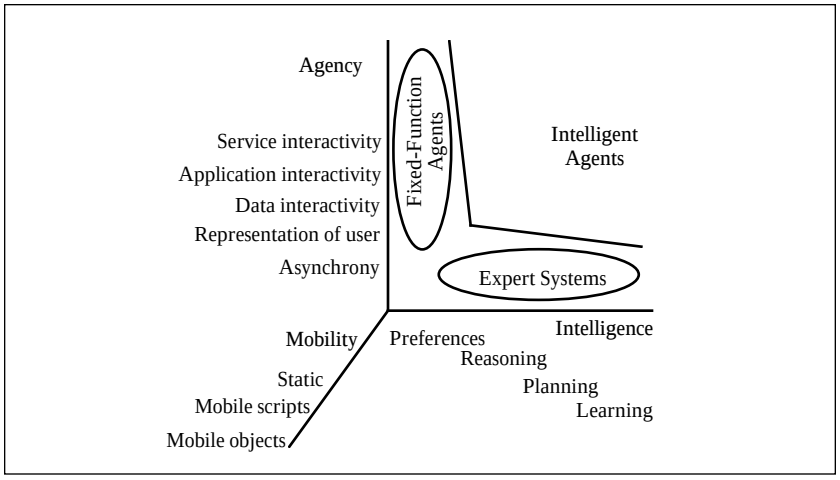


Figure 1. Scope of intelligent agents (Adapted from Gilbert et al. 1995).

capabilities, a *social* agent possesses explicit models of other agents." (Moulin and Chaib-draa 1996, pp. 8-9).

An influential white paper from IBM (Gilbert et al. 1995) described intelligent agents in terms of a space defined by the three dimensions of *agency, intelligence,* and *mobility* (figure 1):

"*Agency* is the degree of autonomy and authority vested in the agent, and can be measured at least qualitatively by the nature of the interaction between the agent and other entities in the system. At a minimum, an agent must run asynchronously. The degree of agency is enhanced if an agent represents a user in some way… A more advanced agent can interact with… data, applications,… services… [or] other agents.

Intelligence is the degree of reasoning and learned behavior: the agent's ability to accept the user's statement of goals and carry out the task delegated to it. At a minimum, there can be some statement of preferences… Higher levels of intelligence include a user model… and reasoning.… Further out on the intelligence scale are systems that *learn* and *adapt* to their environment, both in terms of the user's objectives, and in terms of the resources available to the agent…

Mobility is the degree to which agents themselves travel through the network… *Mobile scripts* may be composed on one machine and shipped to another for execution… *[Mobile objects* are] transported from machine to machine in the middle of execution, and carrying accumulated state data with them."

Nwana (1996) proposes a typology of agents that identifies other dimensions of classification. Agents may thus be classified according to:

- Mobility, as *static* or *mobile*
- Presence of a symbolic reasoning model, as *deliberative* or *reactive*

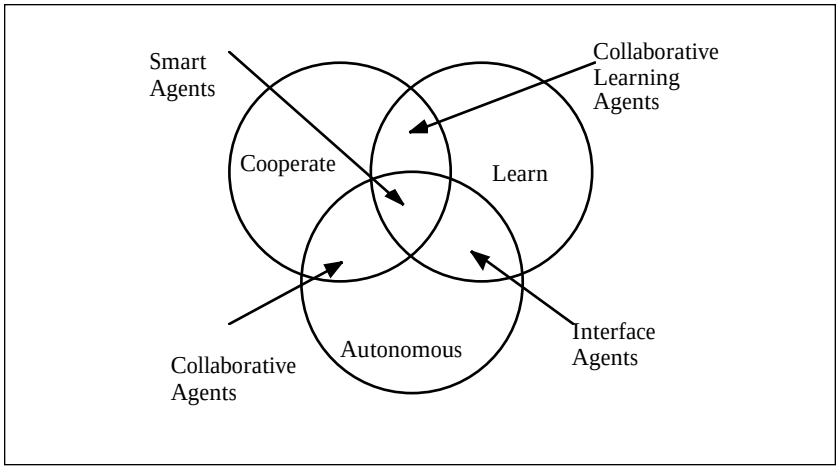


Figure 2. Typology based on Nwana's (Nwana 1996) primary attribute dimension.

- Exhibition of ideal and primary attributes, such as *autonomy, cooperation, learning.* From these characteristics, Nwana derives four agent types: *collaborative, collaborative learning, interface,* and *smart* (see figure 2).
- Roles, as *information* or *Internet*
- Hybrid philosophies, which combine two or more approaches in a single agent
- Secondary attributes, such as versatility, benevolence, veracity, trustworthiness, temporal continuity, ability to fail gracefully, and mentalistic and emotional qualities.

After developing this typology, Nwana goes on to describe ongoing research in seven categories: *collaborative agents, interface agents, mobile agents, information/Internet agents, reactive agents, hybrid agents,* and *smart agents.*

After listing several definitions given by others, Franklin and Graesser (1996) give their own: "an autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." Observing that by this definition even a thermostat could qualify as an agent, they discuss various properties of agents and offer the taxonomy in figure 3 as one that covers most of the examples found in the literature. Below this initial classification, they suggest that agents can be categorized by control structures, environments (e.g., database, file system, network, Internet), language in which they are written, and applications.

Finally, Petrie (1996) discusses the various attempts of researchers to distinguish agents from other types of software. He first notes the difficulties in satisfactorily defining intelligence and autonomy. Then he shows how most of the

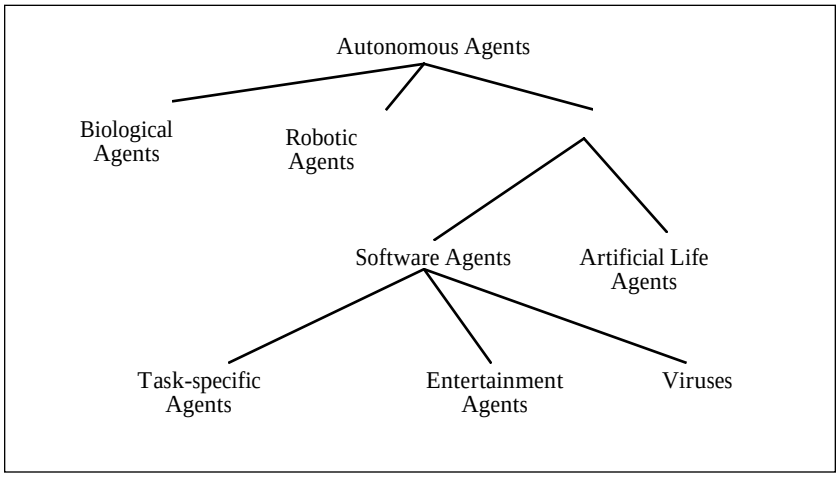


Figure 3. Franklin and Graesser's (1996) agent taxonomy.

current web-based searching and filtering "agents," though useful, "are essentially one-time query answering mechanisms" that are adequately described by the less glamorous computer science term "server." Similarly, "mobile process" would be a less confusing term than "mobile agent" for those Java applets whose only "agent-like" function is to allow processes to run securely on foreign machines. In contrast to these previous attempts to describe a set of unambiguous defining characteristics for agents in general, Petrie argues the case for one specific class: *typed-message agents.* Typed-message agents are distinguished from other types of software by virtue of their ability to communicate as a community using a shared message protocol such as KQML. In the shared message protocol, at least some of the message semantics "are typed and independent of the applications. And semantics of the message protocol necessitate that the transport protocol not be only client/server but rather a peer-to-peer protocol. An individual software module is not an agent at all if it can communicate with the other candidate agents only with a client/server protocol without degradation of the collective task performance."

Time and experience will ultimately determine both the meaning and the longevity of the term "agent." Like many other computing terms in common usage such as "desktop," "mouse," and "broker," it began with a metaphor but will end up denoting concrete software artifacts. As public exposure to useful and technically viable implementations of agent software increases, the term will either come to mean something that everyone understands because they have seen many examples of it, or it will fall into disuse because it describes a concept that is no longer appropriate. What is *unlikely* to disappear are the motivations that have incited the development of agent-based software. These are described in the following section.

Why Software Agents?

While the original work on agents was instigated by researchers intent on studying computational models of distributed intelligence, a new wave of interest has been fueled by two additional concerns of a practical nature: 1) simplifying the complexities of distributed computing and 2) overcoming the limitations of current user interface approaches.[13] Both of these can essentially be seen as a continuation of the trend toward greater abstraction of interfaces to computing services. On the one hand, there is a desire to further abstract the details of hardware, software, and communication patterns by replacing today's program-to-program interfaces with more powerful, general, and uniform agent-to-agent interfaces; on the other hand there is a desire to further abstract the details of the human-to-program interface by delegating to agents the details of specifying and carrying out complex tasks. Grosof (Harrison, Chess, and Kershenbaum 1995) argues that while it is true that point solutions not requiring agents could be devised to address many if not all of the issues raised by such problems, the aggregate advantage of agent technology is that it can address all of them at once.

In the following two subsections, I discuss how agents could be used to address the two main concerns I have mentioned. Following this, I sketch a vision of how "agent-enabled" system architectures of the future could provide an unprecedented level of functionality to people.

Simplifying Distributed Computing

Barriers to Intelligent Interoperability. Over the past several years, Brodie (1989) has frequently discussed the need for *intelligent interoperability* in software systems. He defines the term to mean intelligent cooperation among systems to optimally achieve specified goals. While there is little disagreement that future computing environments will consist of distributed software systems running on multiple heterogeneous platforms, many of today's most common configurations are, for all intents and purposes, disjoint: they do not really communicate or cooperate except in very basic ways (e.g., file transfer, print servers, database queries) (figure 4). The current ubiquity of the Web makes it easy to forget that until the last few years, computer systems that *could* communicate typically relied on proprietary or *ad hoc* interfaces for their particular connection. The current growth in popularity of object-oriented approaches and the development of a few important agreed-upon standards (e.g., TCP/IP, HTTP, IIOP, ODBC) has brought a basic level of encapsulated connectivity to many systems and services. Increasingly, these connections are made asynchronously through message passing, in situations where the advantages of loose coupling in complex cooperating systems can be realized (Mellor 1994; Perrow 1984; Shaw 1996).

We are now in the midst of a shift from the network operating system to In-

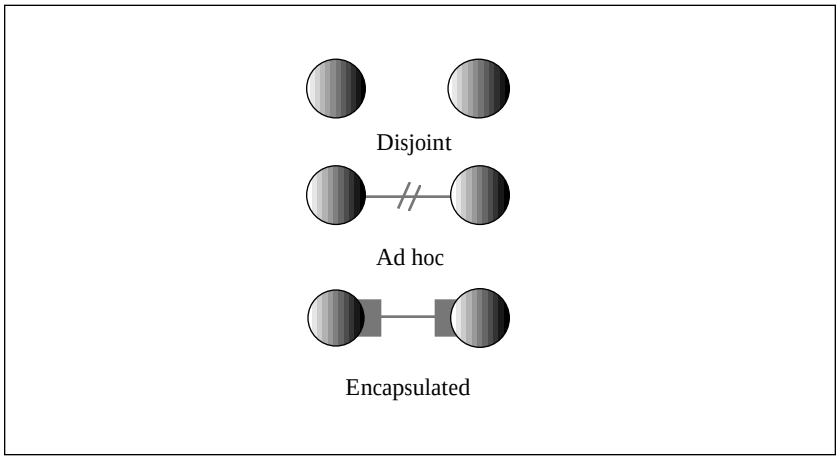


Figure 4. Evolution of system connectivity (Adapted from Brodie 1989).

ternet and intranet-based network computing (Lewis 1996). As this transition takes place, we are seeing the proliferation of operating system-independent interoperable network services such as naming, directory, and security. These, rather than the underlying operating systems, are defining the network, reducing the operating systems to commodities. Lewis (1996) asserts that Netscape is the best example of a vendor focused exclusively on such a goal. Federations of such vendors are defining standards-based operating system-independent services (directory, security, transactions, Web, and so forth), truly universal server-independent clients (Web browsers), and network-based application development support (Java, JavaScript, ActiveX). In such approaches, both the client and server operating systems become little more than a collection of device drivers.

Incorporating Agents as Resource Managers

A higher level of interoperability would require knowledge of the capabilities of each system, so that secure task planning, resource allocation, execution, monitoring, and, possibly, intervention between the systems could take place. To accomplish this, an intelligent agent could function as a global resource manager (figure 5).

Unfortunately, while a single agent might be workable for small networks of systems, such a scheme quickly becomes impractical as the number of cooperating systems grows. The activity of the single agent becomes a bottleneck for the (otherwise distributed) system. A further step toward intelligent interoperability is to embed one or more peer agents within each cooperating system (figure 6). Applications request services through these agents at a higher level corresponding more to user *intentions* than to specific *implementations*, thus providing

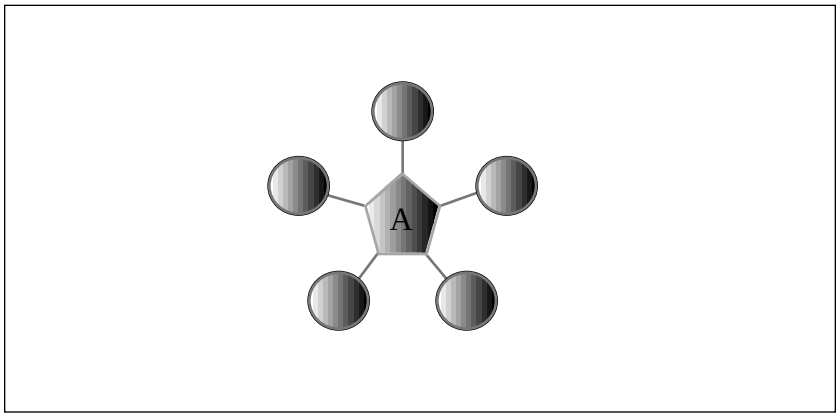


Figure 5. Cooperating systems with single agent as a global planner. Connections represent agent-to-application communication (Adapted from Brodie 1989).

a level of encapsulation at the planning level, analogous to the encapsulation provided at the lower level of basic communications protocols. As agents increasingly evolve from stationary entities to mobile ones, we will see an even more radical redefinition of distributed object computing within corporate networks and on the World Wide Web (Chang and Lange 1996). These scenarios presume, of course, timely agreement on basic standards ensuring agent interoperability (Gardner 1996; Lange 1996; Virdhagriswaran, Osisek, and O'Connor 1995; White 1997).

Overcoming User Interface Problems

Limitations of Direct Manipulation Interface. A distinct but complementary motivation for software agents is in overcoming problems with the current generation of user interface approaches. In the past several years, *direct manipulation* interfaces (Hutchins, Hollan, and Norman 1986; Shneiderman 1983; Shneiderman 1984; Smith, et al. 1982) have become the standard. For many of the most common user tasks, they are a distinct improvement over command-line interfaces. Since direct manipulation requires software objects to be visible, users are constantly informed about the kinds of things they can act upon. If, in addition, the objects have a natural correspondence to real-world or metaphorical counterparts, users can apply previously acquired experience to more quickly learn what the objects can do and how to do it. Many advantages of direct manipulation begin to fade, however, as tasks grow in scale or complexity. For example, anyone who has had much experience with iconic desktop interfaces knows that there are times when sequences of actions would be better automated than directly performed by the user in simple, tedious steps.[14] Several researchers have analyzed the limitations of passive artifact metaphors for com-

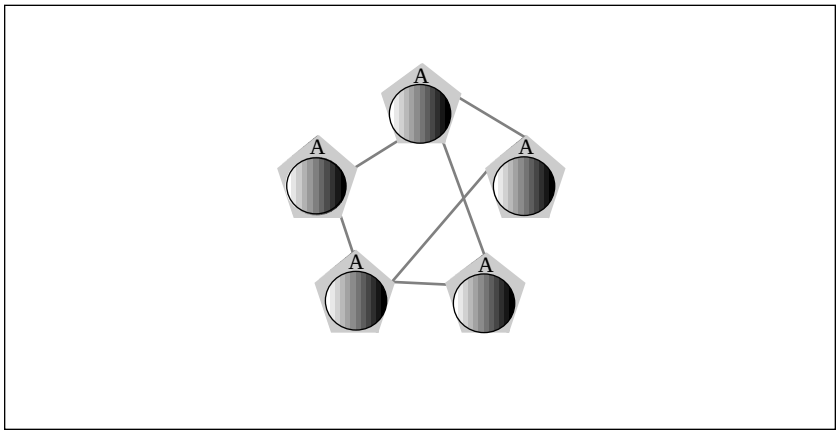


Figure 6. Cooperating systems with distributed agents. Connecting lines represent ongoing agent-to-agent communication (Adapted from Brodie 1989).

plex tasks (diSessa 1986; Erickson 1996; Kay 1990; Whittaker 1990). Among others, people are likely to encounter the following problems:

- *Large search space:* In large distributed systems it is difficult to find what we need through browsing or the use of traditional indexing methods. What is practical and possible for a few hundred items becomes unwieldy and impossible for several thousand.
- *Actions in response to immediate user interaction only:* Sometimes instead of executing an action immediately, we want to schedule it for a specific time in the future. Or, we may want to have software automatically react to system-generated events when we are away from the machine.
- *No composition:* With most direct manipulation interfaces, we cannot easily compose basic actions and objects into higher-level ones.
- *Rigidity:* The consistency that makes passive artifact interfaces predictable and easy-to-learn for simple tasks makes them brittle and untrustworthy for complex ones.
- *Function orientation:* Software is typically organized according to generic software functions rather than the context of the person's task and situation.
- *No improvement of behavior:* Traditional software does not notice or learn from repetitive actions in order to respond with better default behavior.

Indirect Management Using Agents

Researchers and developers are attempting to address these problems by combining the expression of user intention through direct manipulation with the notion of an *indirect management* style of interaction (Kay 1990). In such an ap-

proach, users would no longer be obliged to spell out each action for the computer explicitly; instead, the flexibility and intelligence of software agents would allow them to give general guidelines and forget about the details.

Many of the actions now performed by users could be delegated to various software agents. Thus, in a glimpse of the future, Tesler (1991) imagines the following directives being given by a person to a software agent:

- On what date in February did I record a phone conversation with Sam?
- Make me an appointment at a tire shop that is on my way home and is open after 6 PM.
- Distribute this draft to the rest of the group and let me know when they've read it.
- Whenever a paper is published on fullerene molecules, order a copy for my library.

Later on in the day, Tesler imagines the agent catching up to the person with these follow-up messages:

- You asked me when you last recorded a phone conversation with Sam. It was on February 27. Shall I play the recording?
- You scribbled a note last week that your tires were low. I could get you an appointment for tonight.
- Laszlo has discarded the last four drafts you sent him without reading any of them.
- You have requested papers on fullerene research. Shall I order papers on other organic microclusters as well?

Direct manipulation and indirect management approaches are not mutually exclusive. Interface agent researchers are not out to completely do away with computing as we know it, but more modestly hope that complementing see-and-point interfaces with ask-and-delegate extensions will help reduce required knowledge and simplify necessary actions while maintaining a sufficient level of predictability. Specifically, the use of software agents will eventually help overcome the limitations of passive artifact interfaces in the following ways (table 2):

- *Scalability:* Agents can be equipped with search and filtering capabilities that run in the background to help people explore vast sources of information.
- *Scheduled or event-driven actions:* Agents can be instructed to execute tasks at specific times or automatically "wake up" and react in response to system-generated events.
- *Abstraction and delegation:* Agents can be made extensible and composable in ways that common iconic interface objects cannot. Because we can "communicate" with them, they can share our goals, rather than simply process our commands. They can show us how to do things and tell us what went wrong (Miller and Neches 1987).
- *Flexibility and opportunism:* Because they can be instructed at the level of

Typical Limitations of Direct Manipulation Interfaces	Advantages of Agent-Oriented Approach
Large search space	Scalability
Actions in response to immediate user interaction only	Scheduled or event-driven actions
No composition	Abstraction and delegation
Rigidity	Flexibility and opportunism
Function orientation	Task orientation
No improvement of behavior	Adaptivity

Table 2. Typical limitations of direct manipulation interfaces and advantages of agent-oriented approach.

goals and strategies, agents can find ways to "work around" unforeseen problems and exploit new opportunities as they help solve problems.

- *Task orientation:* Agents can be designed to take the context of the person's tasks and situation into account as they present information and take action.
- *Adaptivity:* Agents can use learning algorithms to continually improve their behavior by noticing recurrent patterns of actions and events.

Toward Agent-Enabled System Architectures

In the future, assistant agents at the user interface and resource-managing agents behind the scenes will increasingly pair up to provide an unprecedented level of functionality to people. A key enabler is the packaging of data and software into components that can provide comprehensive information about themselves at a fine-grain level to the agents that act upon them.

Over time, large undifferentiated data sets will be restructured into smaller elements that are well-described by rich metadata, and complex monolithic applications will be transformed into a dynamic collection of simpler parts with self-describing programming interfaces. Ultimately, all data will reside in a "knowledge soup," where agents assemble and present small bits of information from a variety of data sources on the fly as appropriate to a given context (figure 7) (Neches et al. 1991; Sowa 1990). In such an environment, individuals and groups would no longer be forced to manage a passive collection of disparate documents to get something done. Instead, they would interact with active *knowledge media* (Barrett 1992; Bradshaw et al. 1993b; Brown and Duguid 1996; Glicksman, Weber, and Gruber 1992; Gruber, Tenenbaum, and Weber 1992) that integrate needed resources and actively collaborate with them on their tasks.

Figure 7 illustrates the various roles agents could play in an agent-enabled system architecture. Some could act in the role of intelligent user interface managers,

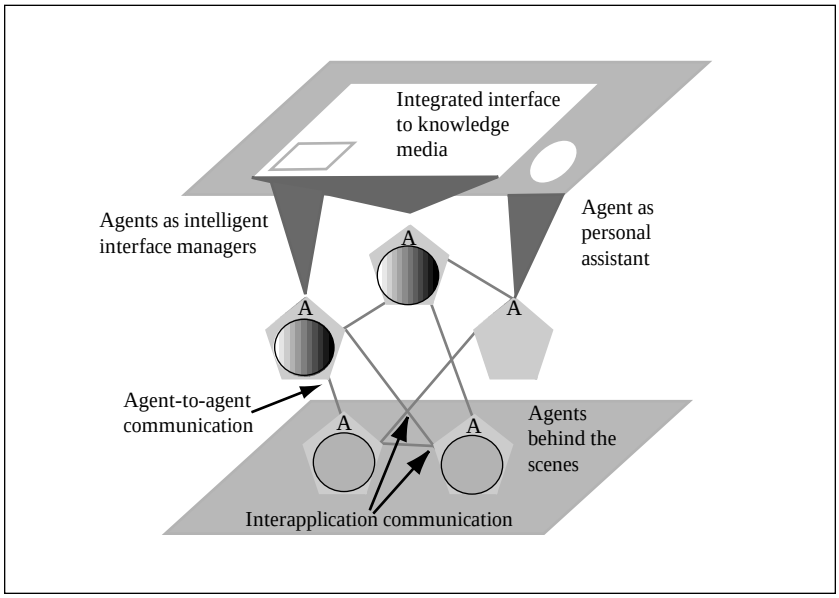


Figure 7 An agent-enabled system architecture.

drawing on the resources of other agents working behind the scenes (Arens et al. 1991; Browne, Totterdell, and Norman 1990; Kay 1990; Neal and Shapiro 1994; Sullivan and Tyler 1991). Such agents would work in concert to help coordinate the selection of the appropriate display modes and representations for the relevant data (Bradshaw and Boose 1992; Johnson et al. 1994), incorporating semantic representations of the knowledge in the documents to enhance navigation and information retrieval (Boy 1992; Bradshaw and Boy 1993; Gruber, Tenenbaum, and Weber 1992; Lethbridge and Skuce 1992; Mathé and Chen 1994). Because the layout and content of the views would be driven by context and configuration models rather than by hand-crafted user-interface code, significant economies could be realized as the data and software components are reused and semi-automatically reconfigured for different settings and purposes. Some agents might be represented explicitly to the user as various types of personal assistants (Maes 1997). Ideally, each software component would be "agent-enabled," however for practical reasons components may at times still rely on traditional interapplication communication mechanisms rather than agent-to-agent protocols.

Overview of the Book

The first subsection summarizes the first set of chapters under the heading of "Agents and the User Experience," which contain introductory pieces authored by proponents (and a critic) of agent technology. The next set, "Agents for

Learning and Intelligent Assistance," describes how agents have been used to enhance learning and provide intelligent assistance to users in situations where direct manipulation interfaces alone are insufficient. The final set, "Agent Communication, Collaboration, and Mobility," details various approaches to agent-oriented programming, agent-to-agent communication, and agent mobility, as well as the use of agents to provide intelligent interoperability between loosely-coupled components of distributed systems.

Agents and the User Experience

How Might People Interact with Agents? Norman's (1997) introductory chapter sets the stage for the first section of the book. "Agents occupy a strange place in the realm of technology," he opens, "leading to much fear, fiction, and extravagant claims." Because the new crop of intelligent agents differ so significantly in their computational power from their predecessors, we need to take into account the social issues no less than the technical ones if our designs are to be acceptable to people:

> "The technical aspect is to devise a computational structure that guarantees that from the technical standpoint, all is under control. This is not an easy task.
>
> The social part of acceptability is to provide reassurance that all is working according to plan… This is [also] a non-trivial task."

The reassurance that all is working according to plan is provided by an understandable and controllable level of feedback about the agent's intentions and actions. We must also think about how to accurately convey the agent's capabilities and limitations so that people are not misled in their expectations. Part of the problem is the natural overenthusiasm of agent researchers; part of the problem is people's tendency to falsely anthropomorphize.[15] Although designers can carefully describe agent capabilities and limitations within accompanying instructional manuals, it is even more important to find clever ways to weave this information naturally and effectively into the agent interface itself.

Safety and privacy are additional concerns. "How does one guard against error, maliciousness (as in the spread of computer viruses), and deliberate intent to pry and probe within one's personal records?" Legal policies to address these issues must be formulatedimmediately, at local, national, and global levels.

A final concern is how to design the appropriate form of interaction between agents and people. For example, how do ordinary people program the agent to do what they want? While programming-by-demonstration or simplified visual or scripting languages have been suggested, none of them seem adequate to specify the kinds of complex tasks envisioned for future intelligent agents.[16]

Since "agents are here to stay," we must learn how to cope with the dangers along with the positive contributions. "None of these negative aspects of agents are inevitable. All can be eliminated or minimized, but only if we

consider these aspects in the design of our intelligent systems."

Agents: From Direct Manipulation to Delegation. In his chapter, Nicholas Negroponte (1997), a longtime champion of agent technology (Negroponte 1970, 1995), extols the virtues of delegation in intelligent interfaces:

> "The best metaphor I can conceive of for a human-computer interface is that of a well-trained English butler. The "agent" answers the phone, recognizes the callers, disturbs you when appropriate, and may even tell a white lie on your behalf. The same agent is well trained in timing… and respectful of idiosyncrasies. People who know the butler enjoy considerable advantage over a total stranger. That is just fine." (Negroponte 1997.)

What will such digital butlers do? They will filter, extract, and present the relevant information from bodies of information larger than we could ordinarily digest on their own. They will act as "digital sisters-in-law," combining their knowledge of information and computing services with intimate knowledge about the person on whose behalf they are acting.

To create agents that are intelligent enough to perform these tasks to our level of satisfaction, we will need to re-open profound and basic questions of intelligence and learning that past AI research has left largely untouched. An understanding of decentralized approaches to intelligence is key: coherence can emerge from the activity of independent agents who coordinate their actions indirectly through shared external influences in their common environment.[17]

User interface design will also be decentralized. Instead of being an effort by professionals to produce the best interface for the masses, it will become an individual affair, driven more by the personalized intelligence of one's local agents than the blanket application of general human-factors knowledge. The agent's long-time familiarity with a person, gained by numerous shared experiences, will be critical to the new beyond-the-desktop metaphor. Though direct manipulation has its place, Negroponte believes that most people would prefer to run their home and office life with a gaggle of well-trained butlers.

Interface Agents: Metaphors with Character. The thesis of Laurel's (1997) chapter is that unabashed anthropomorphism in the design of interface agents is both natural and appropriate:

> "First, this form of representation makes optimal use of our ability to make accurate inferences about how a character is likely to think, decide, and act on the basis of its external traits. This marvelous cognitive shorthand is what makes plays and movies work… Second, the agent as character (whether humanoid, canine, cartoonish, or cybernetic) invites conversational interaction… [without necessarily requiring] elaborate natural language processing… Third, the metaphor of *character* successfully draws our attention to just those qualities that form the essential nature of an agent: responsiveness, competence, accessibility, and the capacity to perform actions on our behalf."

Recognizing the considerable resistance many will have to this idea, she responds

to some of the most common criticisms. First, is the objection to having to face "whining, chatting little irritants" each time you turn on the machine. Laurel notes that the problem is not "agents *per se*, but rather the traits they are assumed to possess." To address this problem, we must allow the traits of agents to be fully user-configurable. Another criticism is the indirection implied by the presence of an agent, "Why should I have to negotiate with some little dip in a bowtie when I know exactly what I want to do?" The answer is that if you know what you want to do and if you want to do it yourself, the agent should quickly get out of your way. Agent-based assistance should be reserved for tedious or complex tasks that you don't want to do yourself, and that you are comfortable entrusting to a software entity. Will people's acquired habit of bossing agents around lead them to treating real people the same way? Laurel argues that this is a real issue, but should not be handled by repression of the dramatic form—rather it should be addressed as an ethical problem for agent designers and the culture at large. Finally, there is the oft-heard criticism that "AI doesn't work." Laurel counters with examples of successful use of AI techniques in well-defined domains. Moreove, she asserts that most agents do not need a full-blown "artificial personality," but can be implemented much more simply.

Laurel concludes with a discussion of key characteristics of interface agents (agency, responsiveness, competence, and accessibility) and of an R&D agenda that includes an appreciation of the contribution of studies of story generation and dramatic character.[18]

Designing Agents as if People Mattered. Erickson (1997) explores the many difficulties surrounding adaptive functionality and the agent metaphor. With respect to adaptive functionality, he describes three design issues raised in a study of users of the DowQuest information retrieval system. In brief, people need to *understand* what happened and why when a system alters its response; they need to be able to *control* the actions of a system, even when it does not always wait for the user's input before it makes a move; and they need to *predict* what will happen, even though the system will change its responses over time. Several approaches to these problems have been suggested, including: providing users with a more accurate model of what is going on, managing overblown expectations of users at the beginning so they are willing to persist long enough to benefit from the system's incremental learning, and constructing a plausible 'fictional' model of what is going on.

Given the potential of the agent metaphor as a possible fiction for portraying system functionality, Erickson examines three strands of research that shed some light on how well this approach might work. Designed to encourage students to explore an interactive encyclopedia, the Guides project allowed researchers to observe the kinds of attributions and the level of emotional engagement people had with stereotypic characters that assisted in navigation. Erickson also reviews the extensive research that Nass and his colleagues have

performed on the tendency of people to use their knowledge of people and social rules to make judgments about computers. Finally, he discusses recent research on the reaction of people to extremely realistic portrayals of agents.

In the final section of the chapter, Erickson contrasts the desktop object and agent conceptual models, and argues that they can be used together in the same interface so long as they are clearly distinguished from one another. Specific computing functionality can be portrayed either as an object or an agent, depending on what is most natural. The desktop metaphor takes advantage of users' previous knowledge that office artifacts are visible, are passive, have locations, and may contain things. "Objects stay where they are: nice, safe predictable things that just sit there and hold things." Ontological knowledge of a different sort comes into play when the agent metaphor is employed. Our common sense knowledge of what agents can do tells us that, unlike typical desktop objects, they can notice things, carry out actions, know and learn things, and go places.[19] "Agents become the repositories for adaptive functionality." The overall conclusion is that research "which focuses on the portrayal of adaptive functionality, rather than on the functionality itself, is a crucial need if we wish to design agents that interact gracefully with their users."

Direct Manipulation Versus Agents: Paths to Predictable, Controllable, and Comprehensible Interfaces. Breaking with the tone of cautious optimism expressed in the preceding chapters, Shneiderman, a longtime advocate of direct manipulation, is troubled by the concept of intelligent interfaces in general:

> "First, such a classification limits the imagination. We should have much greater ambition than to make a computer behave like an intelligent butler or other human agent…
>
> Second, the quality of predictability and control are desirable. If machines are intelligent or adaptive, they may have less of these qualities…
>
> [Third,] I am concerned that if designers are successful in convincing the users that computers are intelligent, then the users will have a reduced sense of responsibility for failures…
>
> Finally,… [m]achines are not people… [and if] you confuse the way you treat machines with the way you treat people… you may end up treating people like machines."[20]

Shneiderman backs up his general concerns with lessons from past disappointments in natural language systems, speech I/O, intelligent computer-assisted instruction, and intelligent talking robots.

Shneiderman observes that agent proponents have not come up with good definitions of what is and is not an agent. "Is a compiler an agent? How about an optimizing compiler? Is a database query an agent? Is the print monitor an agent? Is e-mail delivered by an agent? Is a VCR scheduler an agent?" His examination of the literature reveals six major elements of the agent approach: anthropomorphic presentation, adaptive behavior, acceptance of vague goal specification, gives you what you need, works while you don't, and works

where you aren't. The first three, on closer examination, seem counterproductive, while the last three are good ideas that could be achieved by other means.

The alternative to a vision of computers as intelligent machines is that of predictable and controllable user interfaces, based on direct manipulation of representations of familiar objects. Shneiderman concludes with a description of two examples from his own lab (tree maps and dynamic queries) that show the power of visual, animated interfaces "built on promising strategies like informative and continuous feedback, meaningful control panels, appropriate preference boxes, user-selectable toolbars, rapid menu selection, easy-to-create macros, and comprehensible shortcuts." These, he argues, rather than vague visions of intelligent machines, will allow users to specify computer actions rapidly, accurately, and confidently.

Agents for Learning and Intelligent Assistance

Agents for Information Sharing and Coordination: A History and Some Reflections. The chapter by Malone, Grant, and Lai (1997) reviews more than ten years of seminal work on a series of programs which were intended to allow unsophisticated computer users to create their own cooperative work applications using a set of simple, but powerful, building blocks. The work is based on two key design principles, which each imply a particular kind of humility that should be required of agent designers:

> "1. Don't build computational agents that try to solve complex problems all by themselves. Instead, build systems where the boundary between what the agents do and what the humans do is a flexible one. We call this the principle of *semiformal systems*…
>
> 2. Don't build agents that try to figure out for themselves things that humans could easily tell them. Instead, try to build systems that make it as easy as possible for humans to see and modify the same information and reasoning processes their agents are using. We call this the principle of *radical tailorability*…"

Information Lens, the first program in the series, was a system for intelligent sorting and processing of electronic mail messages. *Object Lens* and *Oval* were successor programs providing much more general and tailorable environments that extended beyond the domain of electronic mail filtering.

The name "Oval" is an acronym for the four key components of the system: objects, views, agents, and links. "By defining and modifying templates for various semi-structured *objects*, users can represent information about people, tasks, products, messages, and many other kinds of information in a form that can be processed intelligently by both people and their computers. By collecting these objects in customizable folders, users can create their own *views* which summarize selected information from the objects. By creating semi-autonomous *agents*, users can specify rules for automatically processing this information in different ways at different times. Finally, *links*, are used for connecting and relating different objects" (Lai and Malone 1992).

The authors describe several different applications that were created to show the power and generality of the Oval approach.[21] All these demonstrate the surprising power that semiformal information processing can provide to people, and lend credence to the claim that people without formal programming skills can be enabled to create agent-based computing environments that suit their individual needs.

Agents that Reduce Work and Information Overload. While the developers of Oval have explored ways to simplify agent authoring, Pattie Maes and her colleagues at MIT have pursued an approach that allows personal assistants to *learn* appropriate behavior from user feedback (Maes 1997). The personal assistant starts out with very little knowledge and over time becomes more experienced, gradually building up a relationship of understanding and trust with the user:

> "[We] believe that the learning approach has several advantages over [end-user programming and knowledge-based approaches]… First, it requires less work from the end-user and application developer. Second, the agent can more easily adapt to the user over time and become customized to individual and organizational preferences and habits. Finally, the approach helps in transferring information, habits and know-how among the different users of a community."

A learning agent acquires its competence from four different sources. First, it can "look over the shoulder" of users as they perform actions. Second, it can learn through direct and indirect feedback from the user. Indirect feedback is provided when the user ignores the agent's suggested action. Third, the agent can learn from user-supplied examples. Finally, the agent can ask advice from other users' agents that have may have more experience with the same task.
Two assumptions determine whether the learning approach is appropriate for a given application:

1. *The application should involve a significant amount of repetitive behavior.* Otherwise, there would be no consistent situation-action patterns for the agent to learn.
2. *Repetitive behavior should be different for different users.* Otherwise, the behavior could be more efficiently hard-coded once and for all in a program, rather than implemented using learning agents.

Maes describes four agent-based applications built using the learning approach: electronic mail handling *(Maxims)*, meeting scheduling,[22] Usenet Netnews filtering *(Newt)*, and recommending books, music or other forms of entertainment *(Ringo)*[23] Through clever user feedback mechanisms and tailoring options, this approach provides a great deal of functionality from the combination of relatively simple mechanisms.

KidSim: Programming Agents without a Programming Language. Like Malone and his colleagues, Smith, Cypher, and Spohrer (1997) have focused their attention on the problem of agent authoring. What is unique to their application, however, is that they are trying to create a general and powerful tool for use by

children. To make this possible, they have adopted a "languageless" approach:

> "We decided that the question is not: what language can we invent that will be easier for people to use? The question is: should we be using a language at all?… We've come to the conclusion that since all previous languages have been unsuccessful…, *language itself is the problem.*.
>
> … [The] graphical user interface eliminated command lines by introducing visual representations for concepts and allowing people to directly manipulate those representations… Today all successful editors on personal computers follow this approach. But most programming environments do not. This is the reason most people have an easier time *editing* than *programming."*

KidSim[24] (now called "Cocoa") is a tool kit where children can build worlds populated by agents that they program themselves by demonstration and direct manipulation. Existing agents (simulation objects) can be modified, and new ones can be defined from scratch. Although agents cannot inherit from one another, they can share elements such as rules. In keeping with the design philosophy of direct manipulation, all elements of the simulation are visible in the interface.

"Languageless" programming is accomplished by combining two ideas: *graphical rewrite rules*, and *programming-by-demonstration.*[25] Graphical rewrite rules define transformations of a region of the game board from one state to another. Programming-by-demonstration is accomplished by letting the child put the system into "record mode" to capture all actions and replay them. A major strength of KidSim is that the results of recording user actions can be shown graphically, rather than as a difficult-to-understand script, as in most other such systems.

The authors describe a series of classroom studies in which children from ages eight to fourteen have used development versions of KidSim. The studies have led to a refinement of many of the concepts in KidSim, and have in turn provided convincing evidence that reasonably complex simulations of this sort can be constructed by very young children. No doubt there are many agent applications for adults that could take advantage of similar principles to make programming accessible.

Lifelike Computer Characters: The Persona Project at Microsoft. While many software agent researchers are content to make agents that are merely "useful," others seek the more ambitious goal of making "complete" agents that are highly visible in the user interface, project the illusion of being aware and intentioned, and are capable of emotions and significant social interaction. The *Persona* project (Ball et al. 1997) was formed to prototype possible interfaces to future computer-based assistants, in this case a "conversational, anthropomorphic computer character that will interact with the user to accept task assignments and report results." To be successful, such assistants will need to support interactive give and take including task negotiation and clarifying questions, understand how and when it is appropriate to interrupt the user with a report or request for

input, and acknowledge the social and emotional impacts of interaction.

The creation of lifelike computer characters requires a wide variety of technologies and skills, including speech recognition, natural language understanding, animation, and speech synthesis. A sophisticated understanding of subtle dialogue mechanisms and social psychology is also essential. To add to this challenge, all the necessary computational machinery to support these technologies must ultimately be able to run with acceptable performance on garden variety computing platforms.

The application chosen for the project described in this chapter involves an animated character (Personal Digital Parrot One, PDP1, or Peedy for short) that acts as a knowledgeable compact disc changer: "The assistant can be queried as to what CDs are available by artist, title or genre, additional information can be obtained about the CDs, and a playlist can be generated." Peedy responds verbally and by doing things to spoken commands in a restricted subset of natural language.

The application relies on two major technology components: reactive animation and natural language. *ReActor* represents a visual scene and accompanying entities such as cameras and lights hierarchically. The most interesting aspect of the animation is its reactivity, i.e., the fact that complex behaviors, including "emotion," can be triggered by user input. The natural language capability relies on the *Whisper* speech recognition module and on a broad-coverage natural language parser.

The creation of such prototypes has allowed Ball and his colleagues to discover and explore many little-understood aspects of human-computer interaction and to point the way toward the creation of increasingly sophisticated lifelike conversational assistants in the future.

Software Agents for Cooperative Learning. Boy's chapter (1997) examines the role of agents in learning technology. He briefly reviews significant trends of the past, including computer-based training, intelligent tutoring systems, interactive learning systems, and cooperative learning systems. Computer-supported cooperative learning (CSCL) builds on the lessons learned from these past approaches to provide an environment where knowledge is exchanged via active electronic documents.

Four requirements guide the design of active documents: 1. providing the *appropriate illusion* that is useful and natural for the user to understand its content; 2. providing *appropriate indexing and linking mechanisms* to connect the document with other relevant documents; 3. providing *adaptivity* so that over time the document becomes increasingly tailored to the information requirements of particular users; and 4. including *dynamic simulations* to enable people to understand aspects of complex situations that cannot be adequately represented using a static medium such as paper.

Software agents for cooperative learning are designed to transform standard electronic documents into active ones. Drawing on extensive past research expe-

rience with the *Situation Recognition and Analytical Reasoning (SRAR)* model and the *knowledge block representation* (Boy 1992; Boy 1991; Boy and Mathé 1993; Mathé and Chen 1994), he defines an agent in the context of this chapter to be "a software entity that can be represented by a knowledge block with an interface metaphor (appearance)."

As an example of an agent-based CSCL system Boy describes ACTIDOC, a prototype environment for active documents that has been applied in the domain of physics instruction. ACTIDOC documents consist of an ordered set of pages containing content and software agents (to make the content active). Each agent contains a name, a context, a set of triggering conditions, a set of internal mechanisms, and a set of interface metaphors. From Schank and Jona's (1991) six learning architectures, Boy derives classes of agents useful in active document design: *case-based learning agent, incidental learning agent, problem-solving agent, video database agent, simulation agent, and suggestive-questions agent.* Additionally he defines the roles of *evaluation, instructor aid,* and *networking agents.* These are illustrated using a physics example that demonstrates one way that agents can be used to make document content come alive.

The M System. Based on Minsky's *Society of Mind* (SOM) theory (Minsky 1986), the M system (Riecken 1997) is designed to provide intelligent assistance in a broad range of tasks through the integration of different reasoning processes *(societies of agents).* The architecture has previously been applied in the domains of music composition and intelligent user interface agents; this paper describes how M assists users of a desktop multimedia conferencing environment to classify and manage metaphorical electronic objects such as documents, ink, images, markers, white boards, copy machines, and staplers.

In the Virtual Meeting Room (VMR) application, participants collaborate using pen-based computers and a telephone:

> "Each user is supported by a personalized assistant, which attempts to recognize and define relationships between domain objects, based on the actions performed by the users and the resulting new states of the world. For example, VMR participants may perform actions on a group of electronic documents such as joining them as a set or annotating them collectively. M attempts to identify all domain objects and classify relationships between various subsets based on their physical properties and relevant user actions."

Within M there are five major reasoning processes, each of which are viewed as individual agents: *spatial, structural, functional, temporal, and causal.* Other more simple agents function as *supporting agents.* Functioning as a set of SOM *memory machines,* these supporting agents represent conceptual knowledge about things like color, shape, and spatial relationships. As an architecture of integrated agents, M dynamically generates, ranks, and modifies simultaneous theories about what is going on in the VMR world. As a faithful implementation of SOM theory, M provides for an I/O system, a spreading activation semantic net-

work (to implement Minsky's K-lines/polynemes), a rule-based system, a scripting system, a blackboard system (to implement Minsky's trans-frames and pronomes), and a history log file system.

To Riecken, an agent is fundamentally a simple, specialized "reasoning" process, whereas an assistant is composed of many "agencies of agents:" "To handle a common sense problem, one would not typically call on an agent—instead, one would want an assistant endowed with the talents of many integrated agents."

Agent Communication, Collaboration, and Mobility

An Overview of Agent-Oriented Programming. *Agent-oriented programming* (AOP) is a term that Shoham (1977) has proposed for the set of activities necessary to create software agents. What he means by 'agent' is "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments." Agent-oriented programming can be thought of as a specialization of object-oriented programming approach, with constraints on what kinds of state-defining parameters, message types, and methods are appropriate. From this perspective, an agent is essentially "an object with an attitude."

An agent's "mental state" consists of components such as beliefs, decisions, capabilities, and obligations. Shoham formally describes the state in an extension of standard epistemic logics, and defines operators for obligation, decision, and capability. *Agent programs* control the behavior and mental state of agents. These programs are executed by an *agent interpreter.* In the spirit of speech act theory, interagent communication is implemented as speech act primitives of various types, such as inform, request, or refrain.

An agent interpreter assures that each agent will iterate through two steps at regular intervals: 1) read the current messages and update its mental state (including beliefs and commitments), and 2) execute the commitments for the current time, possibly resulting in further belief change. Shoham's original agent interpreter, AGENT-0, implements five language elements: *fact statements* ("Smith is an employee of Acme"), *communicative action statements* (inform, request, refrain), *conditional action statements* ("If, at time *t,* you believe that Smith is an employee of Acme, then inform agent A of the fact"), *variables,* and *commitment rules* ("If you receive message *x* while in the mental state *y,* perform action *z*").

The basic concepts described by Shoham have influenced the direction of many other agent researchers. He and his colleagues have continued their investigations on several fronts including mental states, algorithmic issues, the role of agents in digital libraries, and social laws among agents.

KQML as an Agent Communication Language. While Shoham's definition of an agent is built around a formal description of its mental state, other groups of researchers have taken agent communication as their point of departure.[26] In this

chapter, Finin, Labrou and Mayfield (1997) justify such a rationale as follows:

> "The building block for intelligent interaction is knowledge sharing that includes both mutual understanding of knowledge and the communication of that knowledge. The importance of such communication is emphasized by Genesereth, who goes so far as to suggest that an entity is a software agent if and only if it communicates correctly in an agent communication language (Genesereth and Ketchpel 1994). After all, it is hard to picture cyberspace with entities that exist only in isolation; it would go against our perception of a decentralized, interconnected electronic universe."

After an overview of the work of the Knowledge Sharing Effort (KSE) consortium (Neches et al. 1991) to tackle various issues relating to software agents and interoperability, the authors focus on one particular result of the effort: KQML (Knowledge Query Manipulation Language).

The authors suggest seven categories of requirements for an agent communication language:

- *Form*. It should be declarative, syntactically simple, and easily readable by people and programs.
- *Content*. A distinction should be made between the language that expresses communicative acts ("performatives") and the language that conveys the content of the message.
- *Semantics*. The semantics should exhibit those desirable properties expected of the semantics of any other language.
- *Implementation*. The implementation should be efficient, provide a good fit with existing software, hide the details of lower layers, and allow simple agents to implement subsets of the language.
- *Networking*. It should support all important aspects of modern networking technology, and should be independent of transport mechanism.
- *Environment*. It must cope with heterogeneity and dynamism.
- *Reliability*. It must support reliable and secure agent communication.

After a review of the features of KQML, the authors describe how the features of KQML support each of these requirements. The authors conclude by describing various applications of KQML and by giving a comparison with two related approaches: AOP and Telescript.

An Agent-Based Framework for Interoperability. Genesereth (1997) continues the theme of agent communication with his chapter on the role of agents in enabling interoperability, meaning that software created by different developers and at different times works together in seamless manner. He discusses two limitations of current software interoperability technologies: 1. they lack the ability to communicate definitions, theorems, and assumptions that may be needed for one system to communicate effectively with another, and 2. there is no general way of resolving inconsistencies in the use of syntax and vocabulary.

Like Finin and his colleagues, Genesereth has been a major contributor to

the KSE. His *ACL* (agent communication language) draws on three cornerstones of the KSE approach: *vocabularies*, (ontologies) *KIF (Knowledge Interchange Format)*, and *KQML*. The vocabulary of ACL is represented as a sophisticated open-ended dictionary of terms that can be referenced by the cooperating agents and applications.[27] KIF is a particular syntax for first order predicate calculus that provides for a common internal knowledge representation, an "inner" language for agents (Genesereth and Fikes 1992). It was originally developed by Genesereth's group and is currently being refined as part of an ISO standardization effort. In the ACL approach, KQML is viewed as a linguistic layer on top of KIF that allows information about the context (e.g., sender, receiver, time of message history) to be taken into account as part of agent messages. In short, "an ACL message is a KQML expression in which the 'arguments' are terms or sentences in KIF formed from words in the ACL vocabulary" (Genesereth and Ketchpel 1994).

The concept of a *facilitator* is central to ACL. Agents and facilitators are organized into a *federated system*, in which agents surrender their autonomy in exchange for the facilitator's services. Facilitators coordinate the activities of agents and provide other services such as locating other agents by name (white pages) or by capabilities (yellow pages), direct communication, content-based routing, message translation, problem decomposition, and monitoring. Upon startup, an agent initiates an ACL connection to the local facilitator and provides a description of its capabilities. It then sends the facilitator requests when it cannot supply its own needs, and is expected to act to the best of its ability to satisfy the facilitator's requests.

Genesereth describes several examples of applications and summarizes issues where further work is needed. ACL is an important step toward the ambitious long-range vision where "any system (software or hardware) can interoperate with any other system, without the intervention of human users or… programmers."

Agents for Information Gathering. The chapter by Knoblock and Ambite (1977) provides an in-depth example of the use of agents for an important class of problems: information gathering. The SIMS architecture for intelligent information agents is designed to provide:

1. *modularity* in terms of representing an information agent and information sources,
2. *extensibility* in terms of adding new information agents and information sources,
3. *flexibility* in terms of selecting the most appropriate information sources to answer a query,
4. *efficiency* in terms of minimizing the overall execution time for a given query, and
5. *adaptability* in terms of being able to track semantic discrepancies among models of different agents."

Each SIMS information agent provides expertise on a specific topic by drawing upon other information agents and data repositories. "An existing database or program can be turned into a simple information agent by building the appropriate interface code, called a *wrapper,* that will allow it to conform to the conventions of the [particular agent] organization… [Such an] approach greatly simplifies the individual agents since they need to handle only one underlying language. This arrangement makes it possible to scale the network into many agents with access to many different types of information sources." Agents that answer queries but do not originate them are referred to as *data repositories.*

"Each SIMS agent contains a detailed model of its domain of expertise [(an ontology)] and models of the information sources that are available to it. Given an information request, an agent selects an appropriate set of information sources, generates a plan to retrieve and process the data, uses knowledge about information sources to reformulate the plan for efficiency, and executes the plan." KQML is used as the communication language in which messages are transmitted among agents, while Loom (MacGregor 1990) is used as the content language in which queries and responses are formulated.

A learning capability helps agents improve their overall efficiency and accuracy. Three modes of learning are used: caching data that is frequently retrieved or which may be difficult to retrieve, learning about the contents of information sources so as to minimize the cost of retrieval, and analyzing the contents of information sources so as to refine its domain model.

To date, the authors have built information agents that plan and learn in the logistics planning domain. They are continuing to extend the planning and learning capabilities of these agents.

KAoS: Toward an Industrial-Strength Open Agent Architecture. It is ironic that as various sorts of agents are increasingly used to solve problems of software interoperability, we are now faced with the problem of incompatible competing agent frameworks:

> "The current lack of standards and supporting infrastructure has prevented the thing most users of agents in real-world applications most need: *agent interoperability* (Gardner 1996; Virdhagriswaran, Osisek, and O'Connor 1995). A key characteristic of agents is their ability to serve as universal mediators, tying together loosely-coupled, heterogeneous components—the last thing anyone wants is an agent architecture that can accommodate only a single native language and a limited set of proprietary services to which it alone can provide access."

The long-term objective of the KAoS (Knowledgeable Agent-oriented System) agent architecture (Bradshaw et al. 1997) is to address two major limitations of current agent technology: 1. failure to address infrastructure, scalability, and security issues; and 2. problems with the semantics and lack of principled extensibility of agent communication languages such as KQML. The first problem is addressed by taking advantage of the capabilities of commercial distributed object

products (CORBA, DCOM, Java) as a foundation for agent functionality, and supporting collaborative research and standards-based efforts to resolve agent interoperability issues. The second problem is addressed by providing an open agent communication *meta*-architecture in which any number of agent communication languages with their accompanying semantics could be accommodated.

Each KAoS agent contains a *generic agent instance*, which implements as a minimum the basic infrastructure for agent communication. Specific extensions and capabilities can be added to the basic structure and protocols through ordinary object-oriented programming mechanisms. Unlike most agent communication architectures, KAoS explicitly takes into account not only the individual message, but also the various sequences of messages in which it may occur. Shared knowledge about message sequencing conventions *(conversation policies)* enables agents to coordinate frequently recurring interactions of a routine nature simply and predictably. *Suites* provide convenient groupings of conversation policies that support a set of related services (e.g., the *Matchmaker suite*). A starter set of suites is provided in the architecture but can be extended or replaced as required.

The authors experience with KAoS leads them to be "optimistic about the prospects for agent architectures built on open, extensible object frameworks and [they] look forward to the wider availability of interoperable agent implementations that will surely result from continued collaboration."

Communicative Actions for Artificial Agents. Cohen and Levesque's (1997) chapter identifies major issues in the design of languages for interagent communication, with specific application to KQML:

> "[The] authors of KQML have yet to provide a precise semantics for this language, as is customary with programming languages.[28] Without one, agent designers cannot be certain that the interpretation they are giving to a "performative" is in fact the same as the one some other designer intended it to have. Moreover, the lack of a semantics for communication acts leads to a number of confusions in the set of reserved "performatives" supplied. Lastly, designers are left unconstrained and unguided in any attempt to extend the set of communication actions."

In KQML, communicative actions are considered to belong to a specific class of speech acts called "performatives" which, in natural language, are utterances that succeed simply because speakers say or assert they are doing so (e.g., "I hereby bequeath my inheritance to my daughter"). The authors identify three general difficulties with KQML. First, the definitions of the performatives suffer from *ambiguity and vagueness*. Second, there are *misidentified performatives*, that should instead be classes as directives (e.g., requests) or assertives (e.g., informs). Third, there are *missing performatives*, such as the commissives (e.g., promises).

The authors respond to these difficulties with an outline an analysis of rational action upon which their theory of speech acts rests. They then show how the speech acts of requesting and informing can be defined in terms of the primitives from this theory. The implications for future KQML design decisions are

twofold. First, if developers are allowed to extend the set of KQML performatives, they must provide both correct implementations of the directive force of new actions as well as assure that the new actions enter into old and new conversation patterns correctly. Second, if the communication primitives are to be handled independently of the content of the message, developers must not allow any attitude operators in the content (e.g., not permit an agent who says that it requests an act to also say that it does not want the act done).

The authors provide a comparison with other agent communication languages including AOP, Telescript, and their own Open Agent Architecture (OAA) approach (Cohen and Cheyer 1994). Additional work in joint intention theory (Cohen 1994; Cohen and Levesque 1991; Smith and Cohen 1995) is required to clarify how communicative actions function in the initiation of team behavior, and how they may be able to predict the structure of finite-state models of interagent conversations as used in agent architectures such as KAoS.[29]

Mobile Agents. Telescript is an object-oriented remote programming language that is designed to address the problem of interoperability for network services (White 1997). What PostScript did for cross-platform, device-independent documents, Telescript aims to do for cross-platform, network-independent messaging:

> "In Telescript technology, mobile agents *go* to *places*, where they perform tasks on behalf of a user. Agents and places are completely programmable, but they are managed by security features such as *permits*, *authorities*, and *access controls*. Telescript technology is portable, allowing it to be deployed on any platform, over any transport mechanism, and through assorted media—wireline and wireless. Telescript technology can also handle different content types, including text, graphics, animations, live video, and sounds. Telescript technology turns a network into an open platform.[30] Simplified development, portability, and support for rich message content make the technology applicable to a range of communicating applications, from workflow automation to information services and from network management to electronic markets" (General Magic 1994).

Telescript technology allows developers to bundle data and procedures into an agent that will be sent over the network and executed remotely on the server.[31] The Telescript agent carries its own agenda and may travel to several places in succession in order to perform a task. Security for host systems is of paramount concern. The Telescript runtime engine can be set to prevent agents from examining or modifying the memory, file system, or other resources of the computers on which they execute. Moreover, each agent carries securely formatted and encrypted identification tickets that must be checked by the host before running code. The ticket may also carry information about what kinds of tasks the agent is permitted to perform, and the maximum resources it is allowed to expend.

White provides a motivation for mobile agent technology in terms of several example applications. A comprehensive overview of Telescript technologies and programming model and a brief discussion of related work round out the chapter.

Parting Thoughts

Readers may legitimately complain about the idiosyncratic selection of chapters for this book. Significant research topics and important bodies of work have certainly been neglected[32] although I hope that some of this may be rectified in a subsequent volume. What I have tried to provide is convenient access to an initial collection of exemplars illustrating the diversity of problems being addressed today by software agent technology. Despite the fact that the solutions described here will ultimately be replaced by better ones; regardless of whether the term "software agent" survives the next round of computing buzzword evolution, I believe that the kinds of issues raised and lessons learned from our exploration of software agent technology points the way toward the exciting developments of the next millennium.

Acknowledgments

Heartfelt thanks are due to Kathleen Bradshaw and to Ken Ford and Mike Hamilton of AAAI Press, who nurtured this project from the beginning and patiently sustained it to a successful end. I am grateful to the authors of the individual chapters for allowing their contributions to appear in this volume, and for many stimulating discussions. Peter Clark, Jim Hoard, and Ian Angus provided helpful feedback on an earlier draft of this chapter. Significant support for this effort was provided by Boeing management including Cathy Kitto, Ken Neves, and Al Erisman; and by my colleagues in the agent research group: Bob Carpenter, Rob Cranfill, Renia Jeffers, Luis Poblete, Tom Robinson, and Amy Sun. The writing of this chapter was supported in part by grant R01 HS09407 from the Agency for Health Care Policy and Research to the Fred Hutchison Cancer Research Center.

Notes

1. Works by authors such as Schelde (1993), who have chronicled the development of popular notions about androids, humanoids, robots, and science fiction creatures, are a useful starting point for software agent designers wanting to plumb the cultural context of their creations. The chapter "Information beyond computers" in Lubar (1993) provides a useful grand tour of the subject. See Ford, Glymour, and Hayes (1995) for a delightful collection of essays on android epistemology.

2. This is perhaps an overstatement, since researchers with strong roots in artificial life (a-life) and robotics traditions have continued to make significant contributions to our understanding of autonomous agents (Maes 1993; Steels 1995). Although most researchers in robotics have concerned themselves with agents embodied in hardware, some have also made significant contributions in the area of software agents. See Etzioni (1993) for arguments that software presents a no-less-attractive platform than hardware for the investigation of complete agents in real-world environments. Williams and Nayak (1996) describe a software-hardware hybrid agent concept they call immobile robots *(immobots).*

3. For example, see the operational definition proposed by Shoham: "An agent is an entity whose state is *viewed as* consisting of mental components such as beliefs, capabilities, choices, and commitments."

4. With apologies to Oliver Goldsmith (Bartlett and Beck 1980, p. 369:9).

5. Alan Turing (Turing 1950) proposed what was perhaps the first attempt to operationalize a test for machine intelligence using the criterion of human ascription. Research on believable agents (Bates et al. 1994), lifelike computer characters (Ball 1996), and agent-based computer games (Tackett and Benson 1985) carries on in the same tradition, aiming to produce the most realistic multimedia experience of computer-based agents possible. As discovered by organizers of the Loebner Prize Competitions (Epstein 1992) and the AAAI Robot Competitions (Hinkle, Kortenkamp, and Miller 1996; Simmons 1995), one significant challenge in objectively judging results of competitions based on pure ascription and performance measures is that unless the evaluation criteria are well-thought out, agents or robots relying on cheap programming tricks may consistently outperform those who may be demonstrating some more legitimate kind of machine intelligence.

6. Russell and Norvig (1995, p. 821) discuss the fact that while ascribing beliefs, desires, and intentions to agents (the concept of an *intentional stance*) might help us avoid the paradoxes and clashes of intuition, the fact that it is rooted in a relativistic folk psychology can create other sorts of problems. Resnick and Martin (Martin 1988; Resnick and Martin 1990) describe examples of how, in real life, people quite easily and naturally shift between the different kinds of descriptions of designed artifacts (see footnote 11). Erickson (1997), Laurel (1997), and Shneiderman (1997) offer additional perspectives on the consequences of encouraging users to think in terms of agents.

7. Milewski and Lewis (1994) review the organizational psychology and management science literature regarding delegation. They draw implications for agent design, including delegation cost-benefit tradeoffs, kinds of communication required, determinants of trust, performance controls, and differences in personality and culture.

8. "The difference between an automaton and an agent is a somewhat like the difference between a dog and a butler. If you send your dog to buy a copy of the *New York Times* every morning, it will come back with its mouth empty if the news stand happens to have run out one day. In contrast, the butler will probably take the initiative to buy you a copy of the *Washington Post*, since he knows that sometimes you read it instead" (Le Du 1994), my translation.

9. Newquist (1994) gives a similarly-flavored critique of the overhyping of "intelligence" in various products.

10. Shoham goes on to cite the following statement by John McCarthy, who distinguishes between the "legitimacy" of describing mental qualities to machines and its "usefulness" "To ascribe certain *beliefs*, *free will*, *intentions*, *consciousness*, *abilities* or *wants* to a machine or computer program is *legitimate* when such an ascription expresses the same information about the machine that it expresses about a person. It is *useful* when the ascription helps us understand the structure of the machine, its past or future behavior, or how to repair or improve it. It is perhaps never *logically required* even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is *most straightforward* for machines of known structure such as thermostats and computer operating systems, but is *most useful* when applied to entities whose structure is very incompletely known" (McCarthy 1979).

11. Of course, in real life, people quite easily and naturally shift between the different kinds of descriptions. For example, Resnick and Martin report the following about their research with children building LEGO robots: "As students play with artificial creatures,

we are particularly interested in how the students think about the creatures. Do they think of the LEGO creatures as machines, or as animals? In fact, we have found that students (and adults) regard the creatures in many different ways. Sometimes students view their creatures on a *mechanistic* level, examining how one LEGO piece makes another move. At other times, they might shift to the *information* level, exploring how information flows from one electronic brick to another. At still other times, students view the creatures on a *psychological* level, attributing intentionality or personality to the creatures. One creature 'wants' to get to the light. Another creature 'likes' the dark. A third is 'scared' of loud noises.

Sometimes, students will shift rapidly between levels of description. Consider, for example, the comments of Sara, a fifth-grader (Martin 1988). Sara was considering whether her creature would sound a signal when its touch sensor was pushed:

'It depends on whether the machine wants to tell… if we want the machine to tell us… if we tell the machine to tell us.'

Within a span of ten seconds, Sara described the situation in three different ways. First she viewed the machine on a psychological level, focusing on what the machine 'wants.' Then she shifted intentionality to the programmer, and viewed the programmer on a psychological level. Finally, she shifted to a mechanistic explanation, in which the programmer explicitly told the machine what to do.

Which is the correct level? That is a natural, but misleading question. Complex systems can be meaningfully described at many different levels. Which level is 'best' depends on the context: on what you already understand and on what you hope to learn. In certain situations, for certain questions, the mechanistic level is the best. In other situations, for other questions, the psychological level is best. By playing with artificial creatures, students can learn to shift between levels, learning which levels are best for which situations." (Resnick and Martin 1990).

12. Ideally, this would include some notion of episodic memory. Unfortunately, only two major examples of "agents" incorporating episodic memory in the literature easily come to mind: Winograd's (1973) SHRDLU and Vere and Bickmore's (1990) "basic agent." For a thought-provoking look into the consequences of a future where a personal "agent" might become the ultimate cradle-to-grave companion, experiencing and remembering every event of a lifetime, see "The Teddy" chapter in Norman (1992).

13. In his widely cited article "Eye on the Prize" (Nilsson 1995), Nilsson discusses the shift of emphasis in AI from inventing general problem-solving techniques to what he calls *performance programs*, and gives his reasons for believing that there will soon be a reinvigoration of efforts to build programs of "general, humanlike competence."

14. While macro and scripting languages are technically adequate to solve this problem, it seems unlikely that the majority of "end users" will ever want to endure what it takes to become proficient with them: "In the past two decades there have been numerous attempts to develop a language for end users: Basic, Logo, Smalltalk, Pascal, Playground, HyperTalk, Boxer, etc. All have made progress in expanding the number of people who can program. Yet as a percentage of computer users, this number is still abysmally small. Consider children trying to learn programming… We hypothesize that fewer than 10% of there children who are taught programming continue to program after the class ends… Eliot Soloway states, 'My guess is that the number… is less than 1%! Who in their right mind would use those languages—any of them—after a class?'" (Smith, Cypher, and Spohrer 1997). While the software agent perspective does not obviate the need for end-user programming, I believe it has potential as one means of simplifying some of the conceptual barriers that users and developers face in designing and understanding complex systems.

15. Fortunately, people have a lot of experience in judging the limitations of those with whom they communicate: "Sometimes people overstate what the computer can do, but what people are extremely good at is figuring out what they can get away with. Children can size up a substitute teacher in about five minutes" (Kahle 1993). For evidence that developers of intelligent software are no less prone than other people to overestimate the capabilities of their programs, see McDermott (1976).

16. *Automatic programming* is an enterprise with a long history of insatiable requirements and moving expectations. For example, Rich and Waters (1988) remind us that "compared to programming in machine code, assemblers represented a spectacular level of automation. Moreover, FORTRAN was arguably a greater step forward than anything that has happened since. In particular, it dramatically increased the number of scientific end users who could operate computers without having to hire a programmer." Today, no one would call FORTRAN a form of automatic programming, though in 1958 the term was quite appropriate. The intractability of fully-automated, completely-general programming is analogous to the problem of automated knowledge acquisition (Bradshaw et al. 1993a; Ford et al. 1993). As Sowa observes: "Fully automated knowledge acquisition is as difficult as unrestricted natural language understanding. The two problems, in fact, are different aspects of exactly the same problem: the task of building a formal model for some real world system on the basis of informal descriptions in ordinary language. Alan Perlis once made a remark that characterizes that difficulty: *You can't translate informal specifications into formal specifications by any formal algorithm.*" (Sowa 1989).

17. Van de Velde (1995) provides a useful discussion of the three coupling mechanisms which can enable coordination between multiple agents: knowledge-level, symbol-level, and structural. Symbol-level coupling occurs when agents coordinate by exchange of symbol structures (e.g., "messages") and knowledge-level coupling occurs when an agent "rationalizes the behavior of multiple agents by ascribing goals and knowledge to them that, assuming their rational behavior, explains their behavior" (i.e., through taking an intentional stance). Structural coupling, as discussed extensively by Maturana and Varela (1992), occurs when two agents "coordinate without exchange of representation,… by being mutually adapted to the influences that they experience through their common environment… For example, a sidewalk… plays a coordinating role in the behavior of pedestrians and drivers… [and] the coordination of [soccer] players (within and across teams) is mediated primarily by… the ball." Similarly, as Clancey (1993) argues, the usefulness of the blackboard metaphor is that it provides an external representation that regulates the coordination between multiple agents.

18. These latter issues are discussed in more detail in a Laurel's (1991) book *Computers as Theatre*.

19. It is also easy for people to assume less tangible qualities about agents like that they are internally consistent, are rational, act in good faith, can introspect, can cooperate to achieve common goals, and have a persistent mental state.

20. A more blunt criticism of agents is voiced by Jaron Lanier (1996), who writes, "The idea of 'intelligent agents' is both wrong and evil. I also believe that this is an issues of real consequence to the near-term future of culture and society. As the infobahn rears its gargantuan head, the agent question looms as a deciding factor in whether this new beast will be much better than TV, or much worse." See also his extended online debate with Pattie Maes in Lanier and Maes (1996).

21. Several commercial products have subsequently incorporated Ovallike capability, though with less generality and sophistication. These include cooperative work tools and databases for semi-structured information such as Lotus Notes (Greif 1994) and Caere

Pagekeeper, as well as mail readers with rule-based message sorting. Workflow management tools with some of these capabilities have also appeared.

22. For other approaches to defining agents for scheduling and calendar management tasks, see Kautz et al. (1993); Mitchell et al. (1994).

23. Maes has formed Firefly Network, Inc. in order to extend the technology developed in Ringo to the Web. Her *firefly* service uses knowledge about people with similar tastes and interests in music and movies as a means of personalizing its recommendations.

24. A sort of Java for kids.

25. For a similar approach that relies on graphical rewrite rules, see Repenning's (1995) Agentsheets.

26. This is of course a caricature of both approaches: Shoham does not ignore the importance of agent communication in AOP; neither would most agent communication researchers argue that some representation of "mental state" is unnecessary. Davies' (1994) Agent-K language is an attempt to build a hybrid that extends AGENT-0 to use KQML for communication.

27. One well-known tool that has been used to construct such vocabularies is *Ontolingua* (Gruber 1992a, 1992b).

28. Recent efforts to provide a semantic foundation for KQML are described in Labrou (1996) and Labrou and Finin (1994). Another more general approach to agent language semantics is currently under development by Smith and Cohen (1995).

29. Such a strategy parallels the approach of Rosenschein, who designed a compiler that generates finite state machines whose internal states can be proved to correspond to certain logical propositions about the environment (Kaelbling and Rosenschein 1990; Rosenschein 1995).

30. General Magic is working hard to assure that Telescript can take maximum advantage of developments in Internet technology. Its Tabriz AgentWare and Agent Tools products (General Magic 1996) integrate Telescript and Web technologies, and White has proposed a common agent platform intended to enable interoperability between Telescript and other mobile agent technology (White 1996).

31. With respect to the relationship between Telescript, Tabriz, and Java, General Magic writes: "It is important to note that Telescript and Java are complementary, interoperable languages. Telescript agents can set parameters for Java applets and Java applets can call Telescript operations on the server. This interoperability allows developers to create solutions that leverage the power of the two environments: Java can be used to create and manage compelling user experiences, while Tabriz can manage transactions, instructions, events, and processes" (General Magic 1996).

32. Much more, for example, could have been included about the veritable explosion of work on agents and the Internet (Bowman et al. 1994; Cheong 1996; Etzioni and Weld 1995; Weld, Marks, and Bobrow 1995). None of the many applications of agent technology in complex application areas ranging from digital libraries (Paepcke et al. 1996; Wiederhold 1995) to systems management (Benech, Desprats, and Moreau 1996; Rivière, Pell, and Sibilla 1996) to manufacturing (Balasubramanian and Norrie 1995) could be included. We have slighted the whole fields of artificial life (Langton 1995) situated automata (Brooks 1990; Kaelbling and Rosenschein 1991), learning and adaptation (Gaines 1996; Maes 1995; Sen et al. 1996), and large portions of the voluminous literature on DAI and other fields where important related work is taking place.

References

Apple. 1993. *AppleScript Language Guide.* Reading, Mass.: Addison-Wesley.

Arens, Y.; Feiner, S.; Foley, J.; Hovy, E.; John, B.; Neches, R.; Pausch, R.; Schorr, H.; and Swartout, W. 1991. Intelligent User Interfaces, Report ISI/RR-91-288, USC/Information Sciences Institute, Marina del Rey, California.

Balasubramanian, S., and Norrie, D. H. 1995. A Multi-Agent Intelligent Design System Integrating Manufacturing and Shop-Floor Control. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95),* ed. V. Lesser, 3–9. Menlo Park, Calif.: AAAI Press.

Ball, G. 1996. Lifelike Computer Characters (LCC-96) Schedule and Workshop Information. http://www.research.microsoft.com/lcc.htm.

Ball, G.; Ling, D.; Kurlander, D.; Miller, J.; Pugh, D.; Skelly, T.; Stankosky, A.; Thiel, D.; Dantzich, M. V; and Wax, T. 1996. Lifelike Computer Characters: The Persona Project at Microsoft Research. In *Software Agents,* ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Barrett, E. 1992. Sociomedia: An Introduction. In *Sociomedia: Multimedia, Hypermedia, and the Social Construction of Knowledge,* ed. E. Barrett, 1–10. Cambridge, Mass.: MIT Press.

Bartlett, J., and Beck, E. M., eds. 1980. *Familiar Quotations.* Boston, Mass.: Little, Brown.

Bates, J. 1994. The Role of Emotion in Believable Agents. *Communications of the ACM* 37(7): 122–125.

Bates, J., Hayes-Roth, B., Laurel, B., & Nilsson, N. 1994. Papers presented at the AAAI Spring Symposium on Believable Agents, Stanford University, Stanford, Calif.

Benech, D.; Desprats, T.; and Moreau, J.-J. 1996. A Conceptual Approach to the Integration of Agent Technology in System Management. In *Distributed Systems: Operations and Management (DSOM-96).*

Bowman, C. M.; Danzig, P. B.; Manber, U.; and Schwartz, M. F. 1994. Scalable Internet Resource Discovery: Research Problems and Approaches. *Communications of the ACM* 37(8): 98–107, 114.

Boy, G. 1992. Computer-Integrated Documentation. In *Sociomedia: Multimedia, Hypermedia, and the Social Construction of Knowledge,* ed. E. Barrett, 507–531. Cambridge, Mass.: MIT Press.

Boy, G. A. 1991. *Intelligent Assistant Systems.* San Diego, Calif.: Academic Press.

Boy, G. A. 1997. Software Agents for Cooperative Learning. In *Software Agents,* ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Boy, G. A., and Mathé, N. 1993. Operator Assistant Systems: An Experimental Approach Using a Telerobotics Application. In *Knowledge Acquisition as Modeling,* eds. K. M. Ford and J. M. Bradshaw, 271–286. New York: Wiley.

Bradshaw, J. M., and Boose, J. H. 1992. Mediating Representations for Knowledge Acquisition, Boeing Computer Services, Seattle, Washington.

Bradshaw, J. M., and Boy, G. A. 1993. Adaptive Documents, Internal Technical Report, EURISCO.

Bradshaw, J. M., Dutfield, S., Benoit, P., & Woolley, J. D. 1997. KAoS: Toward an industrial-strength generic agent architecture. In *Software Agents,* ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Bradshaw, J. M.; Ford, K. M.; Adams-Webber, J. R.; and Boose, J. H. 1993. Beyond the

Repertory Grid: New Approaches to Constructivist Knowledge-Acquisition Tool Development. In *Knowledge Acquisition as Modeling*, eds. K. M. Ford and J. M. Bradshaw, 287–333. New York: Wiley.

Bradshaw, J. M.; Richards, T.; Fairweather, P.; Buchanan, C.; Guay, R.; Madigan, D.; and Boy, G. A. 1993. New Directions for Computer-Based Training and Performance Support in Aerospace. Paper presented at the Fourth International Conference on Human-Machine Interaction and Artificial Intelligence in Aerospace, 28–30 September, Toulouse, France.

Brodie, M. L. 1989. Future Intelligent Information Systems: AI and Database Technologies Working Together. In *Readings in Artificial Intelligence and Databases*, eds. J. Mylopoulos and M. L. Brodie, 623–642. San Francisco, Calif.: Morgan Kaufmann.

Brooks, R. A. 1990. Elephants Don't Play Chess. *Robotics and Autonomous Systems* 6.

Brown, J. S., and Duguid, P. 1996. The Social Life of Documents. *First Monday* (http://www.firstmonday.dk).

Browne, D.; Totterdell, P.; and Norman, M., eds. 1990. *Adaptive User Interfaces.* San Diego, Calif.: Academic.

Canto, C., and Faliu, O. (n.d.). *The History of the Future: Images of the 21st Century*. Paris: Flammarion.

Chang, D. T., and Lange, D. B. 1996. Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW. In Proceedings of the OOPSLA 96 Workshop "Toward the Integration of WWW and Distributed Object Technology."

Cheong, F.-C. 1996. *Internet Agents: Spiders, Wanderers, Brokers, and Bots.* Indianapolis, Ind.: New Riders.

Clancey, W. J. 1993. The Knowledge Level Reinterpreted: Modeling Socio-Technical Systems. In *Knowledge Acquisition as Modeling*, eds. K. M. Ford and J. M. Bradshaw, 33–50. New York: Wiley.

Cohen, P. R. 1994. Models of Dialogue. In Cognitive Processing for Vision and Voice: Proceedings of the Fourth NEC Research Symposium, ed. T. Ishiguro, 181–203. Philadelphia, Pa.: Society for Industrial and Applied Mathematics.

Cohen, P. R., and Cheyer, A. 1994. An Open Agent Architecture. Paper presented at the AAAI Spring Symposium on Software Agents, 21–23 March, Stanford, California.

Cohen, P. R.; and Levesque, H. 1997. Communicative Actions for Artificial Agents. In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Cohen, P. R., and Levesque, H. J. 1991. Teamwork, Technote 504, SRI International, Menlo Park, California.

Coutaz, J. 1990. *Interfaces Homme Ordinateur: Conception et Réalisation.* Paris: Editions Bordas.

Davies, W. H. E. 1994. AGENT-K: An Integration of AOP and KQML. In Proceedings of the CIKM-94 Workshop on Intelligent Agents, eds. T. Finin and Y. Labrou. http://www.csd.abdn.ac.uk/~pedwards/publs/agentk.html.

Dennett, D. C. 1987. *The Intentional Stance*. Cambridge, Mass.: MIT Press.

diSessa, A. A. 1986. Notes on the Future of Programming: Breaking the Utility Barrier. In *User-Centered System Design*, eds. D. A. Norman and S. W. Draper. Hillsdale, N.J.: Lawrence Erlbaum.

Epstein, R. 1992. The Quest for the Thinking Computer. *AI Magazine* 13(2): 81–95.

Erickson, T. 1996. Designing Agents as If People Mattered. In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Etzioni, O. 1993. Intelligence without robotics. *AI Magazine*(Winter), 7-14.

Etzioni, O., & Weld, D. S. 1995. Intelligent agents on the Internet: Fact, fiction, and forecast. *IEEE Expert*, 10(4), 44-49.

Finin, T., Labrou, Y., & Mayfield, J. 1997. KQML as an agent communication language. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Etzioni, O. 1993. Intelligence without Robots: A Reply to Brooks. *AI Magazine* 14(4): 7–13.

Etzioni, O., and Weld, D. S. 1995. Intelligent Agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert* 10(4): 44–49.

Foner, L. 1993. *What's an Agent, Anyway? A Sociological Case Study*, Agents Memo, 93-01, Media Lab, Massachusetts Institute of Technology.

Ford, K. M.; Glymour, C.; and Hayes, P. J., eds. 1995. *Android Epistemology.* Menlo Park, Calif.: AAAI Press.

Ford, K. M.; Bradshaw, J. M.; Adams-Webber, J. R.; and Agnew, N. M. 1993. Knowledge Acquisition as a Constructive Modeling Activity. In *Knowledge Acquisition as Modeling*, eds. K. M. Ford and J. M. Bradshaw, 9–32. New York: Wiley.

Franklin, S., and Graesser, A. 1996. Is It an Agent or Just a Program? A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages.* New York: Springer-Verlag.

Gaines, B. R. 1997. *The Emergence of Knowledge through Modeling and Management Processes in Societies of Adaptive Agents,* Knowledge Science Institute, University of Calgary. Forthcoming.

Gardner, E. 1996. Standards Hold Key to Unleashing Agents. *Web Week* 5, 29 April.

General Magic. 1996. Tabriz White Paper: Transforming Passive Networks into an Active, Persistent, and Secure Business Advantage, White Paper (http://www.genmagic.com/Tabriz/Whitepapers/tabrizwp.html), General Magic, Mountain View, California.

General Magic. 1994. Telescript Technologies at Heart of Next-Generation Electronic Services, News Release, 6 January, General Magic, Mountain View, California.

Genesereth, M. R. 1997. An Agent-based Framework for Interoperability. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Genesereth, M. R., and Fikes, R. 1992. Knowledge Interchange Format Version 3.0 Reference Manual, Logic Group Report, Logic-92-1, Department of Computer Science, Stanford University.

Genesereth, M. R., and Ketchpel, S. P. 1994. Software Agents. *Communications of the ACM* 37(7): 48–53, 147.

Gilbert, D.; Aparicio, M.; Atkinson, B.; Brady, S.; Ciccarino, J.; Grosof, B.; O'Connor, P.; Osisek, D.; Pritko, S.; Spagna, R.; and Wilson, L. 1995. IBM Intelligent Agent Strategy, IBM Corporation.

Glicksman, J.; Weber, J. C.; and Gruber, T. R. 1992. The NOTE MAIL Project for Computer-Supported Cooperative Mechanical Design. Paper presented at the AAAI-92 Workshop on Design Rationale Capture and Use, San Jose, California, July.

Greif, I. 1994. Desktop Agents in Group-Enabled Products. *Communications of the ACM* 37(7): 100–105.

Gruber, T. R. 1992a. *ONTOLINGUA: A Mechanism to Support Portable Ontologies, Version 3.0*, Technical Report, KSL 91-66, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Gruber, T. R. 1992b. A Translation Approach to Portable Ontology Specifications. Paper presented at the Seventh Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada.

Gruber, T. R.; Tenenbaum, J. M.; and Weber, J. C. 1992. Toward a Knowledge Medium for Collaborative Product Development. In Proceedings of the Second International Conference on Artificial Intelligence in Design, ed. J. S. Gero.

Harrison, C. G.; Chess, D. M.; and Kershenbaum, A. 1995. Mobile Agents: Are They a Good Idea? IBM T. J. Watson Research Center.

Hayes-Roth, B.; Brownston, L.; and Gent, R. V. 1995. Multiagent Collaboration in Directed Improvisation. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, ed. V. Lesser, 148–154. Menlo Park, Calif.: AAAI Press.

Hinkle, D.; Kortenkamp, D.; and Miller, D. 1996. The 1995 Robot Competition and Exhibition. *AI Magazine* 17(1): 31–45.

Hutchins, E. L.; Hollan, J. D.; and Norman, D. A. 1986. Direct Manipulation Interfaces. In *User-Centered System Design*, eds. D. A. Norman and S. W. Draper, 87–124. Hillsdale, N.J.: Lawrence Erlbaum.

Johnson, P.; Feiner, S.; Marks, J.; Maybury, M.; and Moore, J., eds. 1994. Paper presented at the AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems, Stanford, California.

Kaehler, T., and Patterson, D. 1986. A Small Taste of SMALLTALK. *BYTE*, August, 145–159.

Kaelbling, L. P., and Rosenschein, S. J. 1991. Action and Planning in Embedded Agents. In *Designing Autonomous Agents*, eds. P. Maes , 35–48. Cambridge, Mass.: MIT Press.

Kaelbling, L. P., and Rosenschein, S. J. 1990. Action and Planning in Embedded Agents. *Robotics and Autonomous Systems* 6(1–2): 35–48.

Kahle, B. 1993. Interview of Brewster Kahle. *Intertek* 4:15–17.

Kautz, H.; Selman, B.; Coen, M.; Ketchpel, S.; and Ramming, C. 1994. An Experiment in the Design of Software Agents. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 438–443. Menlo Park, Calif.: American Association for Artificial Intelligence.

Kay, A. 1990. User Interface: A Personal View. In *The Art of Human-Computer Interface Design*, ed. B. Laurel, 191–208. Reading, Mass.: Addison-Wesley.

Kay, A. 1984. Computer Software. *Scientific American* 251(3): 53–59.

Knoblock, C. A., & Ambite, J.-L. 1996. Agents for Information Gathering. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Labrou, Y. 1996. Semantics for an Agent Communication Language. Ph.D. diss., Dept. of Computer Science, University of Maryland at Baltimore County.

Labrou, Y., and Finin, T. 1994. A Semantics Approach for KQML—A General-Purpose Communication Language for Software Agents. In Proceedings of the Third International Conference on Information and Knowledge Management, eds. N. R. Adam, B. K. Bhargava, and Y. Yesha, 447–455. New York: Association of Computing Machinery.

Lai, K.-Y., and Malone, T. W. 1992. Oval Version 1.1 User's Guide, Center for Coordination Science, Massachusetts Institute of Technology.

Lange, D. B. 1996. Agent Transfer Protocol ATP/0.1 Draft 4, Tokyo Research Laboratory, IBM Research.

Langton, C. G., ed. 1995. *Artificial Life: An Overview.* Cambridge, Mass.: MIT Press.

Lanier, J. 1996. Agents of Alienation. http://www.voyagerco.com/misc/jaron.html.

Lanier, J., and Maes, P. 1996. Intelligent Humans = Stupid Humans? *Hot Wired*, 15–24 July. http://www.hotwired.com/braintennis/96/29/index0a.html.

Laurel, B. 1991. *Computers as Theatre*. Reading, Mass.: Addison-Wesley.

Laurel, B. 1997. Interface agents: Metaphors with Character. In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Le Du, B. 1994. Issue 1309, 13 mai. Les Agents, des Assistants dotés d'Intelligence. 01 Informatique, p. 13.

Lethbridge, T. C., and Skuce, D. 1992. Beyond Hypertext: Knowledge Management for Technical Documentation. Submitted to SIGDOC '92. Ottawa, Ontario, Canada.

Lewis, J. 1996. NETSCAPE Gets Serious about Infrastructure. The Burton Group.

Lubar, S. 1993. *InfoCulture: The Smithsonian Book of Information and Inventions.* Boston, Mass.: Houghton Mifflin.

McCarthy, J. M. 1979. Ascribing Mental Qualities to Machines, Technical Report, Memo 326, AI Lab, Stanford University.

McDermott, D. 1976. Artificial Intelligence Meets Natural Stupidity. *SIGART Newsletter* 57:4–9.

MacGregor, R. 1990. The Evolving Technology of Classification-Based Knowledge Representation Systems. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, ed. J. F. Sowa, 385–400. San Francisco, Calif.: Morgan Kaufmann.

Maes, P. 1997. Agents that Reduce Work and Information Overload. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Maes, P. 1995. Modeling Adaptive Autonomous Agents. In *Artificial Life: An Overview*, ed. C. G. Langton, 135–162. Cambridge, Mass.: MIT Press.

Maes, P., ed. 1993. *Designing Autonomous Agents.* Cambridge, Mass.: MIT Press.

Maes, P., and Kozierok, R. 1993. Learning Interface Agents. In Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), 459–465. Menlo Park, Calif.: American Association for Artificial Intelligence.

Malone, T. W.; Grant, K. R.; and Lai, K.-Y. 1996. Agents for Information Sharing and Coordination: A History and Some Reflections. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Martin, F. 1988. *Children, Cybernetics, and Programmable Turtles.* Masters Thesis, Media Laboratory, Massachusetts Institute of Technology.

Mathé, N., and Chen, J. 1994. A User-Centered Approach to Adaptive Hypertext Based on an Information Relevance Model. Paper presented at the Fourth International Conference on User Modeling (UM '94), Hyannis, Massachusetts.

Maturana, H. R., and Varela, F. J. 1992. *The Tree of Knowledge: The Biological Roots of Human Understanding* (rev. ed.). Boston: Shambala.

Mellor, P. 1994. CAD: Computer-Aided Disaster. *SOFSEM 94.*

Milewski, A. E., and Lewis, S. M. 1994. Design of Intelligent Agent User Interfaces: Delegation Issues. Technical Report, Oct. 20. AT&T Information Technologies Services.

Miller, J. R., and Neches, R. 1987. Tutorial on Intelligent Interfaces Presented at the Sixth National Conference on Artificial Intelligence, 14–16 July, Seattle, Washington.

Minsky, M. 1986. *The Society of Mind.* New York: Simon & Schuster.

Minsky, M., and Riecken, D. 1994. A Conversation with Marvin Minsky about Agents. *Communications of the ACM* 37(7): 23–29.

Mitchell, T.; Caruana, R.; Freitag, D.; McDermott, J.; and Zabowski, D. 1994. Experience with a Learning Personal Assistant. *Communications of the ACM* 37(7): 81–91.

Moulin, B., and Chaib-draa, B. 1996. An Overview of Distributed Artificial Intelligence. In *Foundations of Distributed Artificial Intelligence*, eds. G. M. P. O'Hare and N. R. Jennings, 3–55. New York: Wiley.

Neal, J. G., and Shapiro, S. C. 1994. Knowledge-Based Multimedia Systems. In *Multimedia System*, ed. J. F. K. Buford, 403–438. Reading, Mass.: Addison-Wesley.

Neches, R.; Fikes, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T.; and Swartout, W. R. 1991. Enabling Technology for Knowledge Sharing. *AI Magazine* 12(3): 36–55.

Negroponte, N. 1997. Agents: From Direct Manipulation to Delegation. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Negroponte, N. 1995. *Being Digital.* New York: Alfred Knopf.

Negroponte, N. 1970. *The Architecture Machine: Towards a More Human Environment*. Cambridge, Mass.: MIT Press.

Newell, A. 1982. The Knowledge Level. *Artificial Intelligence* 18:87–127.

Newquist, H. P. 1994. Intelligence on Demand—Suckers. *AI Expert*, December, 42–43.

Nilsson, N. J. 1995. Eye on the Prize. *AI Magazine* 16(2): 9–17.

Norman, D. A. 1997. How Might People Interact with Agents? In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Norman, D. A. 1992. *Turn Signals Are the Facial Expressions of Automobiles.* Reading, Mass.: Addison-Wesley.

Nwana, H. S. 1996. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3): 205-244.

Paepcke, A.; Cousins, S. B.; Garcia-Molina, H.; Hassan, S. W.; Ketchpel, S. P.; Röscheisen, M.; and Winograd, T. 1996. Using Distributed Objects for Digital Library Interoperability. *IEEE Computer*, May, 61–68.

Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technologies.* New York: Basic.

Petrie, C. J. 1996. Agent-Based Engineering, the Web, and Intelligence. *IEEE Expert*, 11(6): 24-29.

Repenning, A. 1995. Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules. Paper presented at Visual Languages, Darmstadt, Germany.

Resnick, M., and Martin, F. 1990. Children and Artificial Life, E&L Memo, 10, Media Laboratory, Massachusetts Institute of Technology.

Rich, C., and Waters, R. C. 1988. Automatic Programming: Myths and Prospects. *IEEE Computer* 21(8): 40–51.

Riecken, D. 1997. The M System. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Rivière, A.-I.; Pell, A.; and Sibilla, M. 1996. Network Management Information: Integration Solution for Models Interoperability, Technical Report, Hewlett-Packard Laboratories.

Rosenschein, S. J. 1985. Formal Theories of Knowledge in AI and Robotics. *New Generation Computing* 3(4): 345–357.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. New York: Prentice-Hall.

Ryan, B. 1991. DYNABOOK Revisited with Alan Kay. *BYTE*, February, 203–208.

Schank, R. C., and Jona, H. Y. 1991. Empowering the Student: New Perspectives on the Design of Teaching Systems. *The Journal of the Learning Sciences* 1(1).

Schelde, P. 1993. *Androids, Humanoids, and Other Science Fiction Monsters.* New York: New York University Press.

Sen, S.; Hogg, T.; Rosenschein, J.; Grefenstette, J.; Huhns, M.; and Subramanian, D., eds. 1996. Adaptation, Coevolution, and Learning in Multiagent Systems: Papers from the 1996 AAAI Symposium. Technical Report SS-96-01. Menlo Park, Calif.: AAAI Press.

Sharp, M. 1993. Reactive Agents, Technical Report, Apple Computer, Cupertino, Calif.

Sharp, M. 1992. Principles for Situated Actions in Designing Virtual Realities. Master's thesis, Department of Computer Science, University of Calgary.

Shaw, M. 1996. Some Patterns for Software Architectures. In *Pattern Languages of Program Design*, eds. J. O. Coplien and D. C. Schmidt, 453–462. Reading, Mass.: Addison-Wesley.

Shneiderman, B. 1997. Direct manipulation vs. agents: Paths to predictable, controllable, and comprehensible interfaces. In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Shneiderman, B. 1987. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, Mass.: Addison-Wesley.

Shneiderman, B. 1983. Direct Manipulation: A Step beyond Programming Languages. *IEEE Computer* 16(8): 57–69.

Shoham, Y. 1997. An Overview of Agent-oriented Programming. In *Software Agents*, ed J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Shoham, Y. 1993. Agent-Oriented Programming. *Artificial Intelligence* 60(1): 51–92.

Simmons, R. 1995. The 1994 AAAI Robot Competition and Exhibition. *AI Magazine* 16(2): 19–30.

Singh, M. P. 1994. *Multiagent Systems: A Theoretical Framework for Intentions, Know-How, and Communication*. Berlin: Springer-Verlag.

Smith, D. C., Cypher, A., & Spohrer, J. 1997. KidSim: Programming Agents Without a Programming Language. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Smith, D. C.; Irby, C.; Kimball, R.; Verplank, W.; and Harslem, E. 1982. Designing the STAR User Interface. *BYTE* 4:242–282.

Smith, I. A., and Cohen, P. R. 1995. Toward a Semantics for a Speech Act–Based Agent Communications Language. In Proceedings of the CIKM Workshop on Intelligent Information Agents, eds. T. Finin and J. Mayfield. New York: Association of Computing Machinery.

Sowa, J. F. 1990. Crystallizing Theories out of Knowledge Soup. In *Intelligent Systems: State of the Art and Future Systems*, eds. Z. W. Ras and M. Zemankova. London: Ellis Horwood.

Sowa, J. F. 1989. Knowledge Acquisition by Teachable Systems. In *EPIA 89, Lecture*

Notes in Artificial Intelligence, eds. J. P. Martins and E. M. Morgado, 381–396. Berlin: Springer-Verlag.

Steels, L. 1995. The Artificial Life Roots of Artificial Intelligence. In *Artificial Life: An Overview*, ed. C. G. Langton, 75–110. Cambridge, Mass.: MIT Press.

Sullivan, J. W., and Tyler, S. W., eds. 1991. *Intelligent User Interfaces.* New York: Association of Computing Machinery.

Tackett, W. A., and Benson, S. 1985. Real AI for Real Games: In *Technical Tutorial and Design Practice*, 467–486.

Tesler, L. G. 1991. Networked Computers in the 1990s. *Scientific American*, September, 86–93.

Turing, A. M. 1950. Computing Machinery and Intelligence. *Mind* 59(236): 433–460.

Van de Velde, W. 1995. Cognitive Architectures—From Knowledge Level to Structural Coupling. In *The Biology and Technology of Intelligent Autonomous Agents*, ed. L. Steels, 197–221. Berlin: Springer Verlag.

Vere, S., and Bickmore, T. 1990. A Basic Agent. *Computational Intelligence* 6:41–60.

Virdhagriswaran, S.; Osisek, D.; and O'Connor, P. 1995. Standardizing Agent Technology. *ACM Standards View*. In press.

Weld, D.; Marks, J.; and Bobrow, D. G. 1995. The Role of Intelligent Systems in the National Information Infrastructure. *AI Magazine* 16(3): 45–64.

White, J. 1997. A Common Agent Platform, http://www.genmagic.com/Internet/Cap/w3c-paper.htm, General Magic, Inc., Sunnyvale, California.

White, J. 1997. Mobile Agents. In *Software Agents*, ed. J. M. Bradshaw. Menlo Park, Calif.: AAAI Press.

Whittaker, S. 1990. Next-Generation Interfaces. Paper presented at the AAAI Spring Symposium on Knowledge-Based Human-Computer Interaction, Stanford, California, March.

Wiederhold, G. 1995. Digital Libraries, Value, and Productivity, Stanford University.

Wiederhold, G. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, March, 38–49.

Wiederhold, G. 1989. The Architecture of Future Information Systems, Technical Report, Computer Science Department, Stanford University.

Williams, B. C., and Nayak, P. P. 1996. Immobile Robots: AI in the New Millennium. *AI Magazine* 17(3): 17–35.

Winograd, T. 1973. A Procedural Model of Language Understanding. In *Computer Models of Thought and Language*, eds. R. Schank and K. Colby, 249–266. New York: Freeman.

Wooldridge, M. J., and Jennings, N. R. 1995. Agent Theories, Architectures, and Languages: A Survey. In *Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, eds. M. J. Wooldridge and N. R. Jennings, 1–39. Berlin: Springer-Verlag.

Section One

Agents & the User Experience

Agent Communication Languages for Information-Centric Agent Communities[1][2]

Marian Nodine and Damith Chandrasekara
InfoSleuth Group, Microelectronics and Computer Technology Corporation
{nodine, damith}@mcc.com

Abstract

As the complexity and application scope of agent-based systems increases, the requirements placed on Agent Communication Languages have also increased with it. Currently-available ACLs focus on agent-based systems in the domain of knowledge agents. Therefore, they lack certain facilities required to implement large, complex, and robust information-centric agent systems. These facilities are required for efficient information transfer, use of multimedia information, and data security. Furthermore, information-gathering agents tend to execute long-running and/or complex tasks, and require ACL support for managing tasks and conversations. Lastly, information agents (and other types of agents) may require more flexible agent-level management and control, specifically in the areas of mobile resources and/or intermittently-connected users.

In this paper we attempt to discuss the shortcomings of existing ACLs and provide specific solutions to address them. The proposed ACL architecture is based on issues that we have encountered with the InfoSleuth system at MCC and other related work.

1. Introduction

Agent-based systems are groups of agents that work together as a single system to integrate their functionality. They consist of a group or groups of agents that interoperate, cooperating to execute large tasks in a distributed manner. The individual agents are encapsulated, semi-autonomous processes that execute on a computer network, offering their services to other agents or other processes. Each agent is a specialist in a particular task or subtask. To execute a larger, more complex task, an agent-based system composes a solution to the task from the different services offered by the individual agents in its system. Naturally, a key piece in this picture is the necessity for the agents to communicate among themselves to coordinate the execution of these complex tasks.

Agent communication languages (ACLs) allow agents to communicate with each other about how to partition these tasks, and to specify the responsibilities of the individual agents that are invoked. Proposed agent

[1] This material is based upon work supported by DARPA under Contract No. N66001-97-C-8500.
[2] Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

communication language standards include FIPA [FIPA] and various flavors of KQML [KQML-Classic], [KQML-93], [KQML-97], [Lab96] [KQML- Lite]. These proposals are oriented towards *speech act* theory. Speech acts are utterances that perform some action or request some specification. An ACL message is a representation of a speech act, and thus provides guidelines as to the interpretation of its contents. This facilitates openness by providing a structure on which patterns of discourse can be defined.

As an ACL, the FIPA proposal is oriented towards agents sharing knowledge, and attempts to define each speech act in the context of some belief system. In an information-oriented system, beliefs are very cumbersome, not well understood by the community, and thus often ignored. For example, FIPA defines the information passing performatives **`tell`** and **`inform`**, where the specifications require the agents to use modal logic to correctly interpret the contents. However, many agents, including most information-centric agents, generally do not implement modal logic. These agents need to be able to exchange messages without needing these extensive capabilities.

The other group of "standard" agent communication languages, KQML ("Knowledge Query Manipulation Language"), comes in several distinct versions. Some of these are more amenable to use in information-centric agent systems. The original version, now referred to as "KQML Classic" [KQML-Classic], has a nested message formulation that is particularly flexible and amenable to various uses among information agents. "KQML 93" and its close relative "KQML 97" [Lab96],[KQML-97] have some problematic features. "KQML Lite" [KQML-Lite], now being developed by DARPA, shows some promise of solving the problems of its precursors. However, again, KQML in its various flavors is still somewhat knowledge-centric.

In this paper, we explore issues and requirements that information-centric agents place on an agent communication language. Information-centric agents focus their efforts on collecting, processing, analyzing, and monitoring large amounts of information, and presenting the information at appropriate levels and in appropriate ways to users interested in that information. The issues we address here have grown out of our extensive experiences with InfoSleuth [BBB+97], an information-centric agent system that has been under development for four years and in use for the last two years. We are specifically interested not only in developing an ACL for the information realm, but also one that is useful among the domain of knowledge agents as well. We then take these requirements and show how they practically extend into a reasonable agent communication language. This

language includes not only a set of messages representing the various speech acts, but also a set of conversational paradigms that support the passing of information and knowledge around efficiently and usefully. Of particular importance is the desire not to encumber a message with interpretational constraints such as beliefs or information formats when the receiving agent cannot understand the nuances among these constraints.

2. Motivational System and Issues

InfoSleuth – The Motivational System

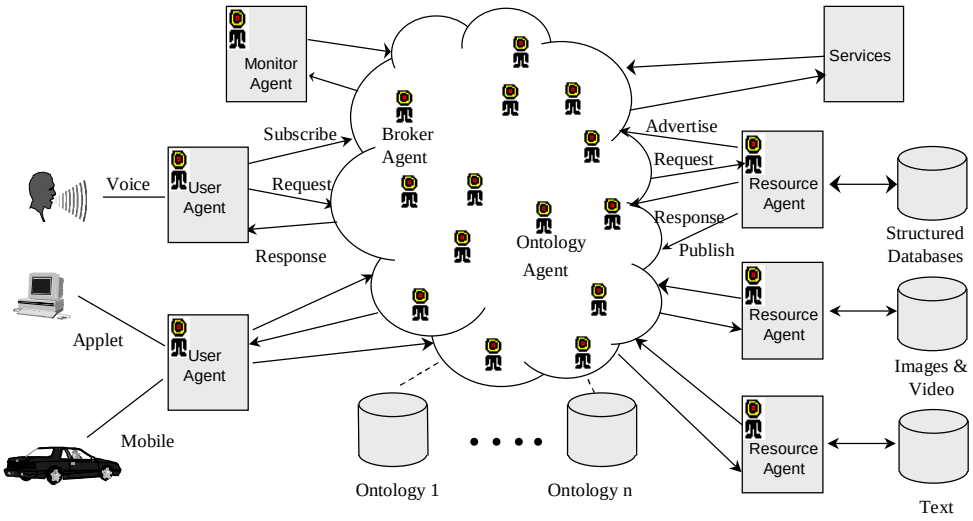


Figure 1. InfoSleuth: Dynamic & Broker-based Agent Architecture

The InfoSleuth model [BBB+97,NPU98] defines a proven framework for loosely collecting agents based on semantic advertisements and dynamically composing agents based on application needs. Figure 1 shows the agent architecture currently defined and deployed in the MCC InfoSleuth project. *User agents*, shown on the left, stand as proxies for individual users or groups of users. Users or other task initiators specify what information they want, over what sort of time span, at what level they want the information abstracted or consolidated to, and may specify how they want to view the returned results. *Resource agents*, on the right, serve as interfaces to external information sources. This information can be stored in files, databases, or text or document collections. Resources can contain any kind of information – structured data, HTML data, image/video data, and semi-structured data. The "cloud" in the middle represents the large and diverse set of agents that work together to connect users with

the information they need. These agents service requests over a set of common ontologies, accessed via the *Ontology agents*. The *Broker agent* serves as a repository for information about agents, and matches requests for services with agents that can service the request. Other agents exist to do *task planning* and *execution*, *query processing*, *data analysis* and *data mining*. All of these agents communicate over a common agent communication language, a variant of KQML, using a common set of query languages and domain ontologies.

A few notable aspects of this architecture, and especially of the agents in the "cloud", are:

- InfoSleuth offers a set of information gathering and monitoring agents. These include: advertisement and brokering, information source wrapping and monitoring, user persistence and representation, data correlation detection across sources, complex query processing and event detection across sources, statistical data analysis and mining, and tracking agent activity in subsets of the agent network.
- Agents advertise their "information gathering and monitoring" capabilities, described using semantic constructs from the InfoSleuth agent capability ontology and a domain ontology. An agent's capabilities can apply to portions of an ontology. In other words, an agent may constrain its capability advertisement to apply to only a select set of concepts, relationships, or instances from a particular domain. These are represented as constraints over this domain ontology.
- In query processing, users formulate information gathering tasks using terms from a domain-specific ontology; the InfoSleuth system dynamically constructs information gathering agent communities, based on brokering and planning principles, to satisfy the task as best as possible.
- In monitoring for information, the user specifies queries over the ontology. The queries which may include relevant abstractions over the information, and may request InfoSleuth to monitor for changes in the queried information over time. This allows the user to view large amounts of information at an appropriate level of abstraction, and to recognize significant changes at the same level of abstraction.

The InfoSleuth architecture, as it currently stands, excels at performing information gathering and monitoring in networks of dynamically appearing and continually changing information sources. Its current Java implementation provides for agent portability across several platforms.

We note in passing that there exist many other information-centric agent systems. SIMS [AKS96], Warren/Retsina [DSW96, DS97], Infomaster [GGKS95], TSIMMIS [GPQR+95], and DISCO [TRV95] all to some

degree are information-centric agent systems. The designers and implementers of these systems have also addressed some of the issues that we discuss in this paper.

A Layered Infrastructure

The operation of an agent system can roughly be broken into four layers, which we will call the *system layer*, the *task layer*, the *conversation layer*, and the *message layer*. These layers together define some of the required functionality for agent interoperation.

The system layer governs how the agent-based system works – how the agents are put together to accomplish their tasks, how they monitor and control their internal operation. For example, one important aspect of the system layer in InfoSleuth is that it has Broker Agents that serve as repositories for information about agents, and that can reason over requests for services and match agents with requested services.

The task layer processes user- or external process- initiated requests into the agent system. The task layer maps individual requests onto specific sub-tasks, and distributes the subtasks among the agents in the system. Either inherent or explicit policies govern how the group of agents cooperate and communicate to complete the task. The structure of these tasks[3] tends to be somewhat abstract, and evolves as the task is being executed.

A conversation begins where an agent requests a specific service from another agent. Modulo issues of forwarding the message, delegating the task, or carbon-copying the response, these conversations are essentially pairwise exchanges between agents. With respect to these conversations, the flow of messages between the agents usually follows some well-defined structure.

Messages are the individual packets of information passed between the agents during conversations. Of all the layers that we have discussed, this has been the most thoroughly studied in the ACL literature.

3. A General Plea

Having worked with both information-centric agents and with interoperability issues with other agent systems, e.g. SIMS [AKS96], we would like first to issue a plea for an ACL with the following properties:

---

[3] Note that some systems refer to "conversations" as occurring at this layer. We will maintain the term "conversation" as we use it in InfoSleuth, to mean agent conversations as defined in the next paragraph.

- A minimal, but standard set of speech acts that are applicable across multiple domain areas.
- An underlying message structure that is easy to parse, interpret, and process while being extensible.
- A definitive "basic" set of standardized conversation structures.
- Support for practical operational issues in systems such as robustness, monitoring, security, and self-organization.
- No assumptions about the underlying transport mechanism should be made, though the specification may give suggestions in order to encourage interoperability among agent-based systems.

For example, with respect to the speech acts, current standards utilize a set of speech acts that is more semantically rich than they need to be for general operation. Thus, we advocate for pushing some of the nuances that are currently distinguished at the speech act level down to properties within the message. This has two advantages. One is that we can define a standard set of conversations over the speech acts more easily, and these conversations can be used for both knowledge- and information-centric agent systems. This facilitates interoperation with other agent systems, even with agent systems of different types. The second is that it is easier to alter the message-specific properties to support more flexible types of service requests, without impacting the basic underlying conversation structure. As we move towards having at least some standard core, these issues would become more critical.

With respect to the second point, we have found with our KQML experience that representing a message as an unordered property list makes it harder to parse and process. Furthermore the nesting of certain types of message such as standby can make the message hard to interpret. Nested messages are hard to interpret because they can have conflicts between the basic properties at the different levels of nesting in the message, and it is difficult (or at least cumbersome) to resolve those conflicts. Thus, any straightforward message specification should minimize the number of nested message types.

With respect to the problem of parsing and processing the unordered property lists, we would like to refer the reader back to the layered infrastructure defined earlier. Different layers within the infrastructure only need to examine specific parts of the incoming message. The remainder of the message is left to the "application" part of the agent, i.e., the part that may be oriented towards information or knowledge or some other domain type. For efficiency and intelligent operation, the part of an agent that implements a specific layer needs only to examine

certain properties of the message – for instance, in KQML, the conversation layer focuses on the `:reply-with` and `:in-reply-to` fields, and the message type, when matching an incoming or outgoing message with its appropriate conversation. We propose the following properties, which are sorted by layer. Properties at any layer should appear before properties in any lower layers. The specifics of these properties will become clear later in the paper.

Layer	Fields
Message layer	
Conversation layer	:sender, :receiver, :conversation-id, :sequence-number, :reply-to :cc-to :flow-policy
Task layer	:task-id, :query-pedigree, :result-pedigree, :result-explanation, :result-annotation, : query-effort, :query-context, :ask-policy, :reply-with-estimate, :reply-out-of-band, :locator
System layer	:content, :language, :ontology, :code
Application	:content, :language, :ontology, other optional properties[4]

We have already, in another paper [NU97], argued for a standardized "base" set of conversation policies, again to extend the standard and to facilitate interoperation with other agent systems. In the current context, these "base" conversations should, at least at the level of exchanges of speech acts, be independent of whether the agents are knowledge-centric, information-centric, a mix of both, or operate in some other domain type.

Lastly we point out that both FIPA and KQML focus on information transmission between the "application" parts of the agents, but there is also an infrastructure to all agents devoted to at least some of the functions of managing transport and connectivity, message processing, conversation management, and task-level management. Often, agents' infrastructures need to exchange messages among themselves – classic examples of this in KQML include the **`transport-address`** performative at the level of managing connectivity, and the **`next`** performative at the level of flow control in conversation management. In InfoSleuth, we also have requirements for additional facilities to use; including those for exchanging monitoring data and for pinging other agents to see if they are up. Some of these facilities overlap with the semantics of the speech acts, but are addressed invisibly of the agent application. An ACL needs to define how to implement these facilities within context.

[4] In InfoSleuth, we include `:ask-policy`, `:reply-policy`, `:reply-summary` here.

4. Message layer

The message layer is where individual speech acts are represented as messages. Messages represent the smallest interaction that can occur between two agents.

4.1 Requirements

Multimedia information presents an additional challenge beyond that of structured data, in that individual data items are rather bulky and irregular. Text documents often contain special characters that overlap with the set of special characters used as delimiters in the ACLs message types. Video and image data are represented as byte streams, and are not ASCII data at all. None of these characteristics can be efficiently handled in a string-oriented ACL such as KQML.

If the ACL uses special characters such as parentheses or quotes within its message structure, the presence of those special characters in the data should be isolated from the message. Also, the ACL should allow data that is not formatted as ASCII strings within its messages.

Sometimes it is very inefficient to pass multimedia data within the message. In most cases the best solution is to return the data out-of-band. This approach will be discussed in the Task Layer section. Still the ACL should not put any restriction as to what can be contained within the content of a message.

In summary, we have the following requirements at the message layer:

A1 ***The ACL should not impose any requirements on the representation of the "content" part of its messages.***

A2 ***ACL replies must include the ability to be self-describing, i.e., to include a description of their own format and contents.***

4.2 Suggested ACL Support

The requirement that there be no restrictions on the representation of the content (Requirement A1) indicates that there are certain parts of the message that should not even be looked at except by the application. Therefore, we suggest that all messages be represented as property lists, where each value of a property is represented by a length-encoded string. The string is preceded by its length (e.g., KQML represents strings in a format like

"6#'string" where "6" is the length and "string" is the value). Thus, the parser can ignore the content of a string that is a property value. Alternatively, the ACL could allow the contents to be sent as a URL, and retrieved out-of-band by some different transport mechanism such as ftp or http, which is better-suited towards handling files.

With respect to messages being self-describing (Requirement A2), this is important either when sending data on the sender's own initiative (tell), or at the request of the receiver (reply). The self-description also may apply to contents that are included in the message (either tell or reply), or to contents that are available only out-of-band (tell-summary or reply-summary). The self-description would also apply to message encryption and compression schemes.

5. Conversation layer

The conversation layer is the layer of the system delivering message interactions between agents. The flow of messages in a conversation follows well-defined structures.

5.1 Requirements

General Conversation Issues

Conversation Identification is an issue that is usually ignored in ACLs and often causes problems. For example KQML supports the chaining of messages using the `:reply-with` and `:in-reply-to` fields, which are predicated on a request-response model. Therefore one would have trouble chaining together messages in conversations that do not follow that model. Suppose an agent were to submit a query involving a large computation. At some point, the agent wishes to cancel the query or possibly amend the query to provide additional restrictions because it is taking too long. However, since no response has been received there is no incoming message to chain the **`discard`** message to. Therefore it is evident that it is necessary to be able to specify which conversation a message belongs to.

Conversations with multiple messages add complexity and versatility to communication between agents but it creates the problem of message ordering. This problem is important when dealing with multimedia content since the order of the messages is very important. The data in these streams may either be naturally ordered (e.g.

Multimedia) in the underlying system, or it may be ordered by its "rank" or match to the underlying query. The ACL need to provide information for assembling the messages into the proper sequence if required. KQML messages, for instance, do not support any message parameters that would inherently aid in the reordering of messages on the other end. The ordering issues also hold true when exchanging knowledge, as the interpretation of new knowledge is often dependent on the current knowledge. Thus, for example, two interdependent **`tell`**s may have a different effect on the knowledge base if they are received in different orders.

Therefore we have the following general requirements.

B1. The messages in the ACL should provide information as to which conversation it pertains to.

B2. The messages in the ACL should provide the information needed to ensure that the messages are received in the sequence in which they are sent.

B3. The ACL should provide information about the origin and destination of the messages.

B4. The ACL should provide support to notify in the event of an error and acknowledge receipt of messages.

Multi-Agent Conversations

One of the main problems faced by an information-centric agent system is that there is a lot of duplication of data. The reason for this is that the conversations used by these systems are usually pair-wise. For information-centric agents this is a serious problem since frequently large intermediate results are passed between agents, and usually all these agents do is forward it to the next agent in the chain. If conversations were not restricted to two agents, then it would be possible for agents to communicate more efficiently. Also conversations would not require additional parameters to support multicasting, delegating, and forwarding. Even forum type discussions between agents would be possible.

Service-oriented conversations are not necessarily pairwise. For instance, one agent may make a subscription request over a set of information, then other agents may themselves subscribe to the same set. In this case, the agent system (for efficiency) may multicast the responses to all of the subscribers. This is one example of a situation where we have found the need for multicasting.

An agent may receive a request that it wishes to delegate in entirety to another agent. In this case although the processing agent receives the request from the delegating agent it would send the reply to the agent specified to it

by the delegating agent. Also sometimes an agent might make a request, but want copies of the responses to be forwarded to another agent.

This gives us the following multi-agent conversation requirements

B5. The ACL should support conversations involving more than one agent

B6. An Agent should have the ability to multicast a message to multiple agents.

B7. An Agent should be able to redirect replies to another agent.

B8. An Agent should be able to forward copies of messages to other agents.

Flow Control

In a data-intensive environment, query results frequently are large, involving many possibly ordered results. This is very typical during query processing when exchanging intermediate results. The typical way that this is addressed in the existing ACLs is to use a streaming facility. With this approach, results are returned one at a time to the requesting agent. However, in an information-intensive agent system, we need a different type of streaming facility. In particular, we need to be able to divide the large result into uniform blocks, sending one block per message. We can then pipeline these messages back, using flow control techniques to ensure that the receiver does not get flooded with results. Therefore we have the following flow control requirement.

B9. The ACL should provide underlying support for streaming back data, and for flow control

5.2 Suggested ACL Support

General Conversation Issues

By including a unique conversation identifier field in each message we can solve the issue of matching messages to the conversations (Requirement B1).

Each message can contain a field with the sequence number of the message in the conversation. This would solve the requirement of providing message-ordering information (Requirement B2).

An **`Error`** message would be needed to signal in the event of an error (Requirement B4).

Therefore the following fields are defined to support the above requirements.

Property	Value semantics	Requirement	Message type
:conversation-id	<unique conversation identifier>	B1	All
:sequence-number	<identifier to place message in proper sequence>	B2	All

:sender	<agent name>	B3	All
:receiver	<list of agents>	B3	All

Multi-Agent Conversations

To support multi-agent conversations and multicasting (Requirements B5 and B6) one must be able to specify a list of agents in the receiver field of the message.

So support delegation (Requirement B7) we would need an additional field to specify which agent to send the replies to. The agent that is doing the delegation would send the request to the agent that is processing the task. The request would contain a `:reply-to` field telling it where to send the replies. That way the results would be sent directly to the requesting agent rather than being funneled through the delegating agent.

Forwarding (Requirement B8) could be accomplished by send a request with a `:cc-to` field set to the agents that should get copies of the replies.

Property	Value semantics	Requirement	Message type
:receiver	<list of agents>	B5, B6	All
:reply-to	<list of agents>	B7	All
:cc-to	<list of agents>	B8	All

Flow Control

To support flow control (Requirement B9) in the Agent Communication Language we would need additional messages that could be passed between agents to control the data flow between the agents. The requestor also needs to be able to describe the number of tuples it is willing to receive in a single block, and the number of blocks of messages that can be "in transit" at a given time for flow control. Streaming and flow control may be supported by the addition of new message types **`next`**, **`stop`** and **`done`**.

A **`next`** message is an indicator from the receiver to the sender that it can send another reply.

A **`stop`** terminates the stream from the requestor's side, and

A **`done`** indicates that there is no more data to send.

We propose the following properties be associated with such messages:

Property	Value semantics	Requirement	Message type
:flow-policy	<block size, number of outstanding blocks>	B9	Ask

Each reply is of size <block-size>, and <num-out> indicates the number of replies that can be in transit at any given time. The entire result is the concatenation of the contents of all of the replies, in the order they were sent. When Agent Y starts responding, it primes the channel with <num-out> replies, then after that it sends the next response every time it receives a **`next`** message from the receiver.

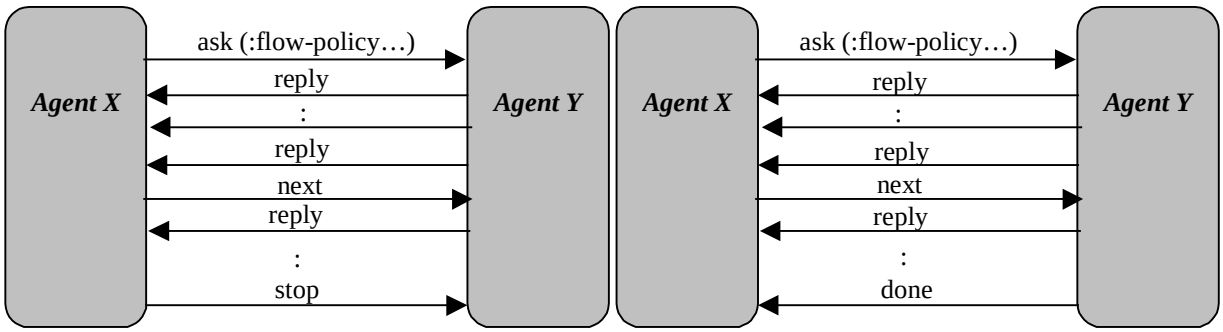


Figure 2. Conversations to support streaming and flow control

6. Task layer

At the task layer, an ACL needs to facilitate the management of the task-related services within the agent. A prime example of where this becomes useful in the data domain is during transaction management.

6.1 Requirements

General Task Issues

At the layer of tasks, an ACL needs to support the ability to relate incoming service request conversations with their corresponding outgoing service conversations easily. To facilitate this, there should be some easy way of matching the incoming conversations with the set of threads accessing outgoing conversations. Messages in the ACL should contain an overall task identifier that is constant for all conversations pertaining to the task.

The task model implied by the ACL should allow the requesting agent to terminate gracefully the computation of a task and the transmission of results at any time, including both before it receives any results and during the middle of the computation and transmission.

The ACL should have support for an agent to refuse service requests from certain agents.

This gives us the following general requirements.

C1. Each task in the system should have its unique identifier that could be added to all messages pertaining to that task.

C2. The ACL should support asserting and querying for information, and replying to queries.

C3. An Agent should be able to terminate a task at any time.

C4. An Agent should be able to refuse service

Explanation of Results

One issue that comes up immediately once changes are allowed in the underlying set of resources is the notion of uncertainty in the returned answer. For instance, if you execute the same query at two separate times, you may get different results because one of the resources accessed during the first query has gone offline, or a new resource was accessed when processing the query the second time. A query that ran once may not return any information the second time because some resource has gone offline. Results may need to contain a result *pedigree* (origin) so that the requesting user can understand where the information came from, either because he wants to judge the quality of the results or because he wants to understand why some information is missing.

Results may be incomplete, or only answer some part of the query. Ideas related to this have been explored, for instance, in the DISCO [TRV95] system. The return messages need to allow for the responding agent to explain the nature of incomplete results and why the query was not completely processed.

Messages containing information should be able to be annotated with support for using or viewing that information. Similar issues hold when returning information or simply telling another agent something. For example, we have already noted that the response may be a summary, locator or estimate. Depending on the type of information, the result also may be annotated with preferred ways of viewing the result; for instance, it may recommend that large amounts of numerical data be displayed as a graph or scatter plot. Also, the user may want to understand where the results of a query came from.

Therefore we have the following explanation of results requirements.

C5. Results may need to contain a result pedigree so that the requesting agent can understand where the information came from.

C6. The Results need to allow the responding agent to explain the nature of incomplete results, and why the query was not completely processed.

C7. Results might need to contain information on how to use or view the information.

Querying

For all aspects of agent interaction relating to data - telling or updating, querying, and responding, there are nuances about how the information is requested or presented that should not necessarily show up at the layer of the message type / speech act. For instance, if you want to subscribe to a set of information and be notified of changes, you may want the notification to happen as soon as the information changes or you may want to know periodically. Furthermore, you may want to receive a fresh copy of all your data, or you may want to receive just the changes.

There are many issues related to the fact that resources may contain overlapping sets of data, also there may be a lot of resources available, containing more information than is really needed by the requester. In this situation, it is helpful to constrain the set of agents accessed, and sometimes to return a "best effort" result as opposed to a complete result.

Certain types of information access do not lend themselves well to having the set of resources change without restriction. For example, a user may wish to browse through a specific set of related information rather than ask specific queries. Also, sometimes a user may wish to formulate a query based on some response to an earlier query. In a data mining application, the user may wish to analyze a constant set of data in different ways. Given that a result may be annotated with the resources where it came from, i.e. its pedigree, then further queries may need to be constrained to access only those resources, or the state of the system at the time the earlier query was made. Thus, the set of results and their pedigree may define a context over which the user can operate.

The querying model should support the return of an estimate before the query is processed. This estimate could be the time the query would take to process or the approximate size of the result, etc. The message should indicate that this is an estimate.

As an alternative to receiving results through conversations, the information can also be returned out-of-band. Here, the direct response to a query may consist of a locator for the information results, and possibly a summary of the results. In InfoSleuth we have explored using out-of-band transmission of bulky, non-ASCII data such as images. With this approach, a locator such as a URL is returned in place of each multimedia item. During query

processing, for instance, this URL can often just be passed along as a part of the result. Any agent interested in actually looking at the item can use the URL to retrieve it out-of-band by some more efficient means, such as ftp, or http.

Thus we have the following query requirements.

C8. An Agent should be able to specify how queries are performed and the type of results that are returned.

C9. An agent should be able to query for a "best effort" result or by amount of data to return as well as querying for a complete result.

C10.An Agent should be able to specify a context over which it can operate.

C11.An Agent should be able to request an estimate before query processing.

C12.An Agent should be able to receive results out of band.

6.2 Suggested ACL support

General Task Issues

To be able to identify which task a particular message belongs (Requirement C1) we would need to define a task identifier field for each message. This field could be named `:task-id`.

Property	Value semantics	Requirement	Message type
Task-id	<unique task identifier>	C1	All

We suggest supporting the following messages to fulfill the above requirements (Requirements C2 - C4).

A **`tell`** can be used by an agent to assert information to another agent.

An **`ask`** can be used by an agent to query another agent.

A **`reply`** is used to reply to a query.

A **`subscribe`** can be used to subscribe to information from an agent.

A **`done`** is used to signal end of results.

An **`end`** is used to terminate a task.

A **`sorry`** is used to signal that the agent cannot or will not provide service.

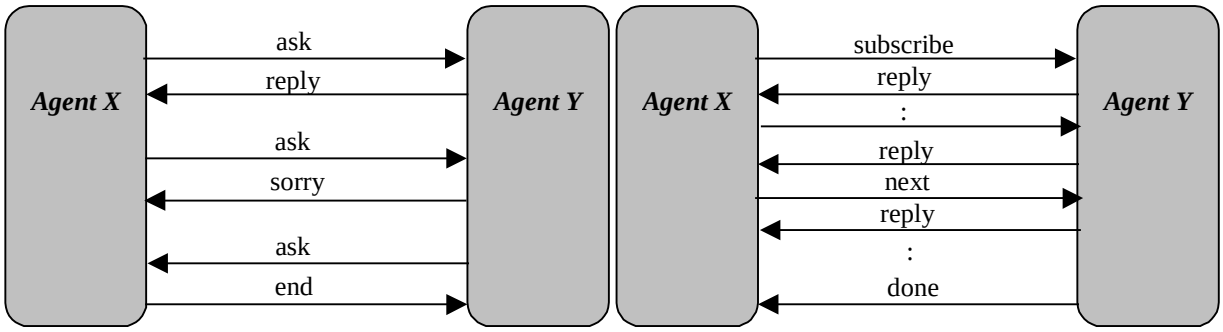


Figure 3. Conversations to support asserting and querying for information, and replying to queries

Explanation of Results

The following fields where defined to support the above mentioned requirements (Requirements C6 and C7).

Propery	Value semantics	Requirement	Message type
:return-pedigree		C5	Ask. Subscribe
:result-pedigree	<list of resources>	C5	Reply
:result-explanation	<explanation>	C6	Reply
:result-annotation	<annotation information>	C7	Reply

Querying

The following fields where defined to support the above mentioned requirements (Requirement C8 – C12).

Property	Value semantics	Requirement	Message type
:ask-policy	<change in data, periodic, all the data, modifications only, etc>	C10	Subscribe
:query-effort	<best effort, complete result>	C8	Ask, Subscribe
:query-context	<context of query>	C9	Ask. Subscribe
:reply-with-estimate	N/A	C11	Ask. Subscribe
:reply-out-of-band	<ftp, http, etc>	C12	Ask. Subscribe
:locator	<URL locator>	C12	Reply

7. System Layer

At the system layer, an agent-based system needs to be able to monitor and manage its own internal operation and its interoperation with other agent-based systems. This monitoring and management should be shielded from the "application" part of the agents, but is rather embedded in the agent infrastructure.

7.1 Requirements

Advertisements

An agent must be able to advertise its capabilities in more depth than just advertising its service interface. We have discussed this issue in more detail in [NU96]. This requirement has been particularly strong in the InfoSleuth system, which is designed to adapt to changing availability of agents and resources. When an agent advertises itself, it offers up its services to the agent system. The clearer the specification of this advertisement, the better matchmaking that the broker can do. In InfoSleuth, we use constraint-based specification of advertisements and of queries, for instance. Therefore, our advertisements are specified as a service interface plus constraints over the information content of the agent, its semantic capabilities, and other agent properties.

Secondly, with respect to a changing system, agents may wish to change its advertisement, or even unadvertise itself if it is about to shut down or wishes to disappear for a while. This enables the broker (or equivalent) to keep its agent repository up-to-date.

Thus, we have the following advertisement-specific ACL requirements:

D1. Advertisements should not be required to use a specific content language, but rather should use a language appropriate to the needs of the system.

D2. The ACL should support the ability of an agent to modify or delete its advertisement.

Monitoring of Agent Operation

An agent-based system should allow the option for its agents to monitor themselves, and to provide information about themselves to other agents as appropriate. During normal operation, monitoring information may be collected at different levels of operation – transport, message, conversation, task, and agent. Additionally, error information may be generated at different levels of operation; for instance, the message layer of an agent that received a malformed message may wish to notify the message layer of the sending agent of the problem. Lastly, the infrastructure may need to maintain a current version of the interconnectivity of the system, using mechanisms such as pinging agents to see if they are alive or subscribing to changes in an agent's state, in which case the agent would notify it if it were to go down.

Therefore we have the following monitoring requirements.

D3. The ACL should define how system monitoring and error information should be passed, and how messages containing this information can be distinguished at the appropriate layer of processing.

D4. The ACL should include at the system layer messages used to monitor connectivity and subscribe to changes in connectivity.

7.2 Suggested ACL Support

Advertisement

We suggest supporting three message types at this layer to support the requirements (Requirements D1 and D2).

`Advertise` asserts positive information about an agent, for either when the agent is starting up or advertising for the first time, or when the agent has additional information to advertise.

`Unadvertise` indicates that specific information about the agent is now invalid.

`Unregister` indicates that the agent is shutting down, and should be removed entirely from the broker's repository.

Property	Value semantics	Requirement	Message type
:content	<service interface and agent semantics>	D1, D2	advertise, unadvertise
:language	<the language the content is specified in>	D1, D2	advertise, unadvertise
:ontology	<the service ontology the content is specified over>	D1, D2	advertise, unadvertise

Therefore we then have the following conversations:

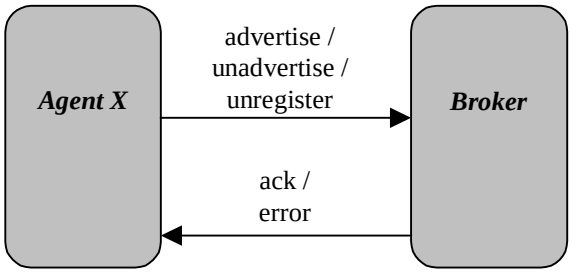


Figure 4. Advertisement-related conversations

Monitoring of Agent Operation

Much of the monitoring and error-communication at the system layer mimics the conversational paradigms that happen at the application level. Therefore, we suggest that monitoring ontologies be defined in conjunction with the ACL, where each monitoring ontology corresponds to some level of processing. Thus, the message layer would have its own ontology that describes the information it can provide (including error types and error codes). Similar ontologies would also be defined for the conversation, task, and system layers. Then, requests and information propagation related to these layers would follow the same message and conversational semantics as the applications use. These messages would actually be handled by the agent's infrastructure at the appropriate layer. Thus, we have the following properties to support monitoring functionality (Requirements D3 and D4).

Property	Value semantics	Requirement	Message type
:content	<message content>	D3	subscribe, ask, tell, reply, error
:ontology	<from {message, conversation, task, system}>	D3	subscribe, ask, tell, reply, error
:language	<language the content is specified in>	D3	subscribe, ask, tell, reply
:code	<error code from ontology>	D3	error

With respect to pinging and connectivity monitoring, we suggest implementing agent ping as an ask message for the remote agent's name over the system ontology, where the name specified in the reply should correlate with the name the requesting agent expects. If an agent is mobile other agents may wish to subscribe to its location.

8. Conclusions

In this paper, we have presented a set of general requirements and approaches towards defining a general ACL that would support information-centric agent systems. Our proposal for a general ACL is not intended to detract from the needs of knowledge-centric agents, but rather to move towards having a standard set of speech acts. These speech acts would be the "common denominator" for both information- and knowledge-centric agents, and could be further tailored to the special needs of both.

As an example, earlier we noted that FIPA supports different speech acts for an agent to tell another agent something, based on what the telling agent believes about itself and what it believes the other agents should do.

An alternative way to represent this would be to carry the beliefs within the message. That is, a **`tell`** would be a basic type of message, and when carrying knowledge the properties of the message would annotate the contents with the beliefs of the sending agent. Other types of belief-oriented annotation might be useful in other circumstances; therefore the extensibility of the ACL would be more focused on extending the properties associated with the messages rather than extending the message types themselves.

With a standard set of basic speech acts defined, we can then extend the ACL to support the different functions required by the conversation, task and system layers of agent-based systems. We have made some specific proposals in this paper towards supporting that functionality. Some issues raised by this architecture have not really been addressed properly in any currently existing ACL, though FIPA [FIPA] is addressing some of these issues. Also, security-related issues specifically are discussed in some detail in [Thi95]. However, issues pertaining to each layer of this architecture must be addressed before we can define a "standard" ACL and consequently move towards an open architecture for agent-based systems as a whole.

Acknowledgements

The authors would like also to thank Brad Perry for his contributions to the ideas in this paper.

References

[AKS96] Y. Arens, C. A. Knoblock, and W. Shen, "Query Reformulation for Dynamic Information Integration," *Journal of Intelligent Information Systems*, 1996

[BBB+97] R. Bayardo et.al, "InfoSleuth: Agent-based semantic integration of Information in open and dynamic environments". In *Proceedings of SIGMOD '97*, 1997.

[CORBA] The Object Management Group and X/Open, The Common Object Request Broker: Architecture and Specification, Revision 1.1, John Wiley and Sons, 1992b.

[DS97] K. Decker and K. Sycara, "Intelligent adaptive information agents". To appear in *Journal of Intelligent Information Systems*.

[DSW96] K. Decker, K. Sycara and M. Williamson, "Modeling information agents: Advertisements, organizational roles, and dynamic behavior." In *Working Notes of the AAAI-96 Workshop on "Agent Modeling"*, 1996.

[FIPA] *FIPA 97 Specification*, http://drogo.cseult.stet.it/fipa/spec/fipa98/fipa98.htm, November, 1997.

[FLM97] T. Finin, Y. Labrou and J. Mayfield, "KQML as an agent communication language". In *Software Agents*, J. M. Bradshaw, ed., AAAI Press, 1997.

[GGKS95] D. Geddis, M. Genessereth, A. Keller and N. Singh, "Infomaster: a virtual information system". *In ACM CIKM Intelligent Information Agents Workshop*, 1995.

[GPQR+95] H. Garcia Molina, Y. Papakonstantinou, D. Quass, A. Rajarman, Y. Sagiv, J. Ullman, and J. Widom, "The TSIMMIS Approach to Mediation: Data Models and Languages," *Proceedings of the NGITS (Next Generation Information Technologies and Systems)*, June 1995.

[Ker97] L. Kerschberg, "The role of intelligent software agents in advanced information systems". *In British National Conference on Databases* (BNCOD 97), 1997.

[KQML97] Y. Labrou and T. Finin, "A Proposal for a New KQML Specification", http://www.cs.umbc.edu/kqml/kqmlspec.ps.

[KQML-Classic] T. Finin and G. Wiederhold, "An Overview of KQML: A Knowledge Query and Manipulation Language", 1991. (available through the Stanford University Computer Science Department).

[KQML-Lite] *KQML Lite Specification*, Technical Report ALP-TR/03, March, 1998 (in progress).

[Lab96] Y. Labrou, Semantics for an Agent Communication Language, Doctoral Disseration, UMBC, September, 1996.

[NPU98] M. Nodine, B. Perry, A. Unruh, "Experience with the InfoSleuth Agent Architecture". In *Proceedings of the AAAI-98 Workshop on Software Tools for Developing Agents*, July, 1998 (to appear).

[NU97] M. Nodine and A. Unruh, "Facilitating Open Communication in Agent Systems: the InfoSleuth Architecture". In *Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages*, July, 1997.

[Thi95] C. Thirunavukkarasu, T. Finin and J. Mayfield, "Secret Agents: A Security Architecture for the KQML Agent Communication Language", in *Proceedings of the Intelligent Information Agents Workshop*, held in conjunction with CIKM '95, Baltimore, December 1995.

[TRV95] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling heterogeneous databases and the design of DISCO", *Proceedings of the International Conference of Distributed Computing Systems*, pp. 449-457, 1996.

Mobile Agents

in the Context of Competition and Co-operation

(MAC3)

http://mobility.lboro.ac.uk/MAC3/

Todd Papaioannou and Nelson Minar

Co-chairs

Preface

Mobile Agent research is a rapidly growing field contributing to autonomous software agents and distributed systems. Mobility is both a useful abstraction *and* tool for agent-based system designers; it allows for increased resource efficiency, capability, and robustness. Also, mobile code is becoming an accepted technology for building distributed systems; allowing distributed systems to be more dynamic and flexible.

While the mobile agent field is a promising area of new research, it comes with many challenges. Although mobility greatly expands the potential of agent systems, the issues of increased complexity and managability must be addressed. Mobile agents also incur new security and consistency problems. Finally, mobility is a relatively new tool for system design; we are still in the early stages of exploring what it is best at. The solution to these problems can be attained through diligent research and communication.

This workshop is one entry in the continuing dialog about mobile agent systems. We hope it will be a valuable opportunity for active researchers in the field to meet, present current and forthcoming research, share ideas, and discuss and critique each other's work.

These workshop notes contain seven papers grouped into four themes. *The Argument for Mobile Agents* contains two papers that discuss in detail why mobile agents are useful; these ideas provide context for mobile agents research. *Tools for managing agent systems* are an important facet of making manageable systems; pattern languages are one such useful tool for organizing mobile agent systems. *Applications* are ultimately what justifies any research; this section contains descriptions of two areas for which mobile agents are particularly well suited. Finally, the workshop notes end with two papers that describe *conceptual frameworks* for mobile agent systems; this research is important so that we may understand the complex software systems we are building.

We wish to thank our program committee for all their help and guidance in organizing the workshop, and all contributors. We hope it will be a valuable experience for everyone involved.

Nelson Minar and Todd Papaioannou

Co-chairs of MAC3 at Agents '99, Seattle.

Table of Contents

Workshop Schedule - 1st May

9:00 - 9:15 Welcome, Introductions

9:15 - 10:00 Keynote

"A Research Agenda for Code Mobility"
Gian Petro Picco, Washington University in St. Louis

10:00 - 10:30 Coffee

10:30 - 11:30 The Argument for Mobile Agents

Mobile Code: The Future of the Internet
David Kotz and Robert Gray, Dartmouth College

Foreign Event Handlers to Maintain Information Consistency and System Adequacy
Pierre-Antonie Queloz, University of Geneva

11:30 - Noon Tools for Managing Mobile Agent Systems

A Case for Mobile Agent Patterns
Dwight Deugo (Carelton University) and Michael Weiss (Mitel Corp.)

Noon - 2:00 Lunch

2:00 - 3:00 Mobile Agent Applications

Mobile agents in an electronic auction house
Qianbo Huai and Tuomos Sandholm (Washington University)

A Partitioning Model for Applications in Mobile Environments
Alexander Schill (Dresden University), Albert Held (DaimlerChrysler), Thomas Ziegert (DU), and Thomas Springer (DU)

3:00 - 3:30 Coffee

3:30 - 4:30 Frameworks for Managing & Understanding Mobile Agent Complexity

Economic Markets as a Means of Open Mobile-Agent Systems

Jonathan Bredin, David Kotz, Daneila Rus (Dartmouth College)

Emergent Behavior and Mobile Agents

Tony White, Bernard Pagurek (Carleton University)

4:30 - 5:30 Discussion - How do we advance mobile agent research?

This program is only a rough guide for the day and is open to change.

The Argument for Mobile Agents

Mobile code: The Future of the Internet

David Kotz and Robert S. Gray

Department of Computer Science / Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire 03755

dfk@cs.dartmouth.edu, robert.s.gray@dartmouth.edu

Abstract

Use of the Internet has exploded in recent years with the appearance of the World-Wide Web. In this paper, we show how current technological trends necessarily lead to a system based substantially on mobile code, and in many cases, mobile agents. We discuss several technical and non-technical hurdles along the path to that eventuality. Finally, we predict that, within five years, nearly all major Internet sites will be capable of hosting and willing to host some form of mobile agents.

1 Introduction

Rapidly evolving network and computer technology, coupled with the exponential growth of the services and information available on the Internet, will soon bring us to the point where hundreds of millions of people will have fast, pervasive access to a phenomenal amount of information, through desktop machines at work, school and home, through televisions, phones, pagers, and car dashboards, from anywhere and everywhere. Mobile agents will be an essential tool for allowing such access. Mobile agents are an effective choice for many reasons [LO99], and although not all applications will need mobile agents, many other applications will find mobile agents the most effective implementation technique for all or part of their tasks.

Although current trends in Internet technology and usage lead inevitably to the use of mobile agents, several technical and non-technical hurdles must be addressed along the way. Although these hurdles represent significant challenges, they can be cleared within years, and nearly all major Internet sites will accept mobile agents within five years. The goal of this position paper is to spark discussion about how best to realize this optimistic, but reasonable, vision.

2 Trends

There are several trends affecting Internet technology and activity:

Bandwidth. The telecommunications industry is laying down astonishing amounts of fiber. Although Internet traffic is growing exponentially, the bandwidth soon to be available on the Internet backbone, as well as to many offices and neighborhoods, is immense.

Nonetheless, bandwidth to many end users will remain limited by several technical factors. Many users will still connect via modem, or at best, ADSL over the old copper loop. Many other users will connect via low-bandwidth wireless networks. Most users can expect to see no more than 128 Kbps to 1 Mbps available at their desktop or palmtop, although some asymmetric cable modems may reach 10 Mbps (for downloads) [Gri99, DR99].

Perhaps more importantly, the *gap* between the low-bandwidth "edge" of the network, and the high-bandwidth "backbone" of the network, will increase dramatically as the backbone benefits from increased quality and availability of fiber, while the edge remains limited by the fundamentals of wireless and copper connections. We expect that this trend will continue even as local connections improve past 1 Mbps in the next few years, since backbone bandwidths are improving much faster than local bandwidths.

Mobile devices. One of the hottest areas of growth in the computer industry is portable computing devices. Everything from laptops to palmtops to electronic books, from cars to telephones to pagers, will access Internet services to accomplish user tasks, even if users have no idea that such access is taking place. Typically, these devices will have unreliable, low-bandwidth, high-latency telephone or wireless network connections.

Mobile users. Web-based email services[1] make it clear that users value the ability to access their email from any computer. Web terminals will become commonplace in public spaces, such as cafes, airports, and hotels. Eventually, particularly with the growth in bandwidth, users will have full access to all of their files and applications from any terminal. Despite this, mobile devices will proliferate unchecked, since just as with public phones, Web terminals will never be available *everywhere* that a user might find herself.

Intranets. Organizations are increasingly using Internet protocols, particularly HTTP, to build internal "intranets" for their own distributed-information needs. Since all access to an intranet is managed by a single organization, new technologies can be deployed quickly, since (1) little coordination is needed with outside organizations, and (2) security (*within* the intranet) is of less concern.

Information overload. Internet users are already overwhelmed by the sheer volume of available information, and the problem will get worse as the Internet grows. Search engines, shopbots, portals, collaborative filtering, and email filtering are existing technologies that allow the user to reduce the torrent to a manageable stream, but these technologies are still quite limited.

Customization. Unlike broadcast media, the Internet makes it possible to customize access for each user. Current technologies allow customization at both the client (browser) and the server. Many Web sites include their own site-specific customization features, but the customization is increasingly provided by third-party "proxy" sites.

Proxies. Such proxy sites, which today are most often Web sites such as the various shopbots, interpose between a user and one or more other Internet services. As a means to both reduce information overload and customize service access, proxy sites will become more and more important. In particular, as portable devices become more prevalent, highly specialized proxy sites will be provided to meet the special needs of mobile users.

3 Mobile agents are inevitable

The trends outlined in the previous section inevitably lead to the conclusion that mobile code, and mobile agents, will be a critical near-term part of the Internet. Why? Not because mobile code makes new applications possible, nor because it leads to dramatically better performance than (combinations of) traditional techniques, but rather because it provides a single, general framework in which distributed, information-oriented applications can be implemented efficiently and easily, with the programming burden spread evenly across information, middleware, and client providers. In other words, mobile code gives providers the time and flexibility to provide their users with *more* useful applications, each with *more* useful features. Our full argument roughly follows Figure 1.

Both the amount of information available on the Internet **(a)**, and the number and diversity of its users **(b)**, are growing rapidly. This diverse population of users will not settle for a uniform interface to the information, but will demand personalized presentations and access methods **(c)**. This personalization will range from different presentation formats to complex techniques for searching, filtering and organizing the vast quantities of information **(d)**. Today, such personalization facilities are provided at the information source in a site-

[1] e.g., `http://www.hotmail.com/`.

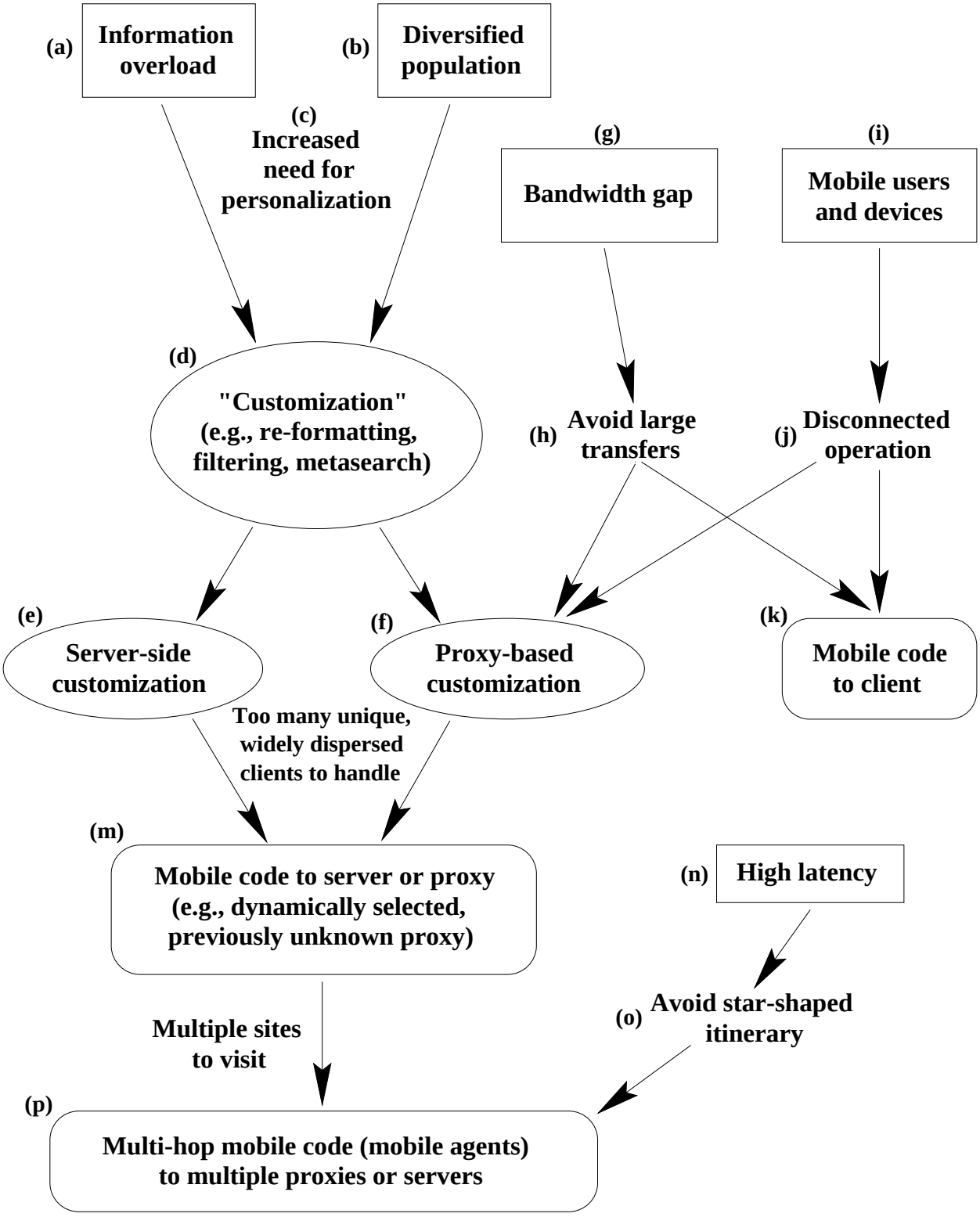


Figure 1: The trends leading to mobile agents

specific manner **(e)**, at a proxy Web site **(f)**,[2] or (occasionally) as client software.[3]

Meanwhile, the network technology will lead to an increased gap in the bandwidth of the core Internet versus the fringes of the Internet **(g)**. Thus, most client hosts will shun large transfers of data **(h)**. That trend encourages the migration of application functionality from clients into proxy sites **(f)**, which are presumably better connected to the core Internet, and need send only the final results over the slower connection to the client. Furthermore, the dramatic availability of core bandwidth will allow these proxy sites to be aggressive in gathering, prefetching, and caching information on behalf of their clients.

Mobile users **(i)** will frequently disconnect from the network, and perhaps connect later at another location with poor bandwidth **(j)**. This tendency again leads to the use of proxies **(f)**. It also encourages application programmers to choose a mobile-code solution to dynamically install the necessary client code **(k)** onto the Web terminal or portable device. Moving code (applets) to the client allows a high level of interaction with the user despite a high-latency, low-bandwidth, or disconnected network.

Ultimately, Web sites and other Internet services will not be able to efficiently provide the full range of customization desired by their clients, and clients will want to use the same information-filtering and -organizing tools across many sites. Moreover, fixed-location, application-specific proxies will become bottlenecks, and as user needs change, may no longer be at the best network location for accessing the proxied services. As a result, customization tools will be specified as software, in the form of mobile code that runs either on the server, or on a dynamically selected proxy site near the server **(m)**. Mobile code is necessary, rather than client-side code, since many customization features (such as information monitoring) do not work if the client is disconnected, has a low-bandwidth connection, or requires frequent communication with the server. Mobile code is beneficial, since servers and proxy sites need provide only a generic execution environment (along with an API that provides programmatic access to their service); the actual customization tools can be written by the services themselves, by third-party middleware developers, and even by the end users.

Finally, many clients will wish to send mobile code to multiple information sites as part of a single task. Although there will be applications for which the mobile code can be sent in parallel, many tasks require a sequence of subtasks, each at a different site. To avoid latency **(n)**, the application programmer will often want to avoid a "star-shaped graph" **(o)** where mobile code goes out to the first site and sends its results back to the client or proxy, the same or different piece of mobile code goes out to the second site, and so on, and the programmer will always want to be able to select the best migration strategy for the task and current network conditions. In other words, the mobile code must be able to hop sequentially through multiple sites; such multi-hop mobile code is a mobile agent.

4 Technical hurdles

There are several technical hurdles that must be cleared before mobile agents can be widely used.

Performance and scalability. Current mobile-agent systems save network latency and bandwidth at the expense of higher loads on the service machines, since agents are often written in a (relatively) slow interpreted language for portability and security reasons, and since the agents must be injected into an appropriate execution environment upon arrival. Thus, in the absence of network disconnections, mobile agents (especially those that need to perform only a few operations against each resource) often take longer to accomplish a task than more traditional implementations, since the time savings from avoiding intermediate network traffic is currently less than the time penalties from slower execution and the migration overhead. Fortunately, significant progress has been made on just-in-time compilation (most notably for Java), software fault isolation, and other techniques [MMBC97], which allow mobile code to execute nearly as fast as natively compiled code. In addition, research groups are now actively exploring ways to reduce migration overhead. Together, these efforts should lead to a system in which accepting and executing a mobile agent involves only slightly more load than if the service machine had provided the agent's functionality as a built-in, natively compiled procedure.

[2] e.g., `http://www.metacrawler.com/`.

[3] e.g., Apple's "Sherlock" meta-search tool.

Portability and standardization. Nearly all mobile-agent systems allow a program to move freely among heterogeneous machines, e.g., the code is compiled into some platform-independent representation such as Java bytecodes, and then either compiled into native code upon its arrival at the target machine or executed inside an interpreter. For mobile agents to be widely used, however, the code must be portable *across mobile-code systems*, since it is unreasonable to expect that the computing community will settle on a single mobile-code system. Making code portable across systems will require a significant standardization effort. The OMG MASIF standard is an initial step, but addresses only cross-system communication and administration [MBB+98], leading to a situation in which an agent can not migrate to the desired machine, but instead only to a nearby machine that is running the "right" agent system. The mobile-agent community must take the next step of standardizing on some specific execution environment(s) (such as a particular virtual machine), as well as on the format in which the code and state of a migrating agent are encoded.

Security. It is possible now to deploy a mobile-agent system that adequately protects a machine against malicious agents [Vig98]. Numerous challenges remain, however: (1) protecting the machines without artificially limiting agent access rights;[4] (2) protecting an agent from malicious machines; and (3) protecting groups of machines that are not under single administrative control. An inadequate solution to any of these three problems will severely limit the use of mobile agents in a truly open environment such as the Internet. Fortunately, groups are now exploring many new techniques, each of which addresses (or partially addresses) one of the three problems (e.g., agents paying for resource usage with electronic cash, which allows them to live and propagate only as long as their cash supply holds out). Although many technical advances (and user-education efforts) must be made before these three problems are solved adequately for *all* Internet applications, current work is promising enough that, within five years, mobile-agent systems will be secure enough for *many* applications.

5 Non-technical hurdles

Once the technical challenges have been met, there remain several non-technical issues that may deter the widespread adoption of mobile-agent technology. Internet sites must have a strong motivation to overcome inertia, justify the cost of upgrading their systems, and adopt the technology. While the technological arguments above are convincing, they are not sufficient for most site administrators. In the end, the technology will be installed only if it provides substantial improvements to the end-user's experience: more useful applications, each with fast access to information, support for disconnected operation, and other important features.

Lack of a killer application. The most important hurdle is that there is no "killer" application for mobile agents. The "mobile agent" paradigm is in many respects a new and powerful programming paradigm, and its use leads to faster performance in many cases. Nonetheless, most particular applications can be implemented just as cleanly and efficiently with a traditional technique, although different techniques would be used for different applications. Thus, the advantages of mobile agents are modest when any particular application is considered in isolation. Instead, researchers must present a set of applications and argue that the *entire set* can be implemented with much less effort (and with that effort spread across many different programming groups). At a minimum, making such an argument demands that the mobile-agent community actively support anyone who is writing a high-quality survey of mobile-agent applications, since no one group will be able to implement a sufficient number of applications. Once a clear quantitative argument is made, a few major Internet services can be convinced to open their sites to mobile agents, since they will recognize that agents will lead to more applications based around their services and hence more users. From there, more Internet services will follow.

Getting ahead of the evolutionary path. It is unlikely that any Internet service will be willing to jump directly from existing client-server systems to full mobile-agent systems. Researchers must provide a clear evolutionary path from current systems to mobile-agent systems. In particular, although full mobile-agent

[4] Many mobile-agent systems reduce an agent's access rights when it arrives from a machine that is not trusted, even if it was launched from a trusted user at a trusted site. The concern is that the agent may have been maliciously modified at the untrusted site.

systems involve all the same research issues (and more) as more restricted mobile-code systems, researchers must be careful to demonstrate that the switch to mobile agents can be made incrementally.

For example, "applets", mobile code that migrates from server to client for better interaction with the user, are in common use, and the associated commercial technology is improving rapidly (e.g., faster Java virtual machines with just-in-time compilation). From applets, the next step is proxy sites that accept mobile code sent from a mobile client. In all likelihood, such proxies will be first provided by existing Internet service providers (ISPs). Since the sole function of the proxy sites will be to host mobile code, and since the ISPs will receive direct payment for the proxy service (in the form of user subscriptions, although not likely at a fixed rate), the ISPs will be willing to accept the perceived security risks of mobile code. Once mobile-code security is further tested on proxy sites, the services themselves will start to accept "servlets", mobile code sent from the client directly to the server (or from the proxy to the server).[5] Once servlets become widely used, and as researchers address the issue of protecting mobile code from malicious servers, services will start to accept mobile agents.

Another critical evolutionary path is the migration of agent technology from intranets to the Internet. Mobile-code technologies will appear first in the relatively safe intranet environment, particularly intranets that are built on high-latency networks such as a WAN or a wireless network for mobile computers. For example, a large company, particularly one with a mobile workforce, might find mobile agents the most convenient way to provide its employees with a wide range of access to its internal databases. Intranets tend to be early adopters of new (useful) technology, because their administrators have more control over the intranet than over the Internet; that control means that security is less of a concern, and wide deployment of agent support services can be encouraged. As the technologies mature in intranets, site administrators will become comfortable with them, and their practicality, safety and potential uses will become clear. Then they will find their way into the Internet.

Revenue and image. A final important hurdle is the problem of revenue flow and commercial image. For example, although it is not yet clear whether advertising is a viable economic foundation for Web sites, many Web sites earn money solely from advertisements. If these sites allow mobile agents to easily access the content of the site, the number of human visits to the Web pages will presumably decrease, and the advertisements will not be seen. How, then, will the site earn revenue? Similarly, when users are accessing a service with a front-end backed by mobile agents, the distinction between the service and the front-end agents starts to blur. Since the agents will likely be provided by middleware developers, the Internet service will no longer have complete control over its image. A poorly implemented agent may lead to a negative view of the *service*, even though the service is blameless. We believe, however, that mobile agents can be deployed in the near-term in many applications where the existing services do not rely on advertising; in the long-term, both the Internet and mobile-agent communities will need to explore different revenue models.

6 Conclusion

There is a strong case for the use of mobile agents in many Internet applications. Moreover, there is a clear evolutionary path that will take us from current technology to widespread use of mobile code and agents within the next five years. Once several technical challenges have been met, and a few pioneering sites install mobile-agent technology, use of mobile agents will expand rapidly.

7 Acknowledgments

Many thanks to the Office of Naval Research (ONR), the Air Force Office of Scientific Research (AFOSR), the Department of Defense (DoD), and the Defense Advanced Research Projects Agency (DARPA) for their financial support: ONR contract N00014-95-1-1204, AFOSR/DoD contract F49620-97-1-03821, and DARPA

[5] Like applets, and unlike agents, servlets are mobile code but not mobile processes. The code is moved from client to server, starts execution, and later ends execution on the same machine. It cannot migrate further once it starts executing.

contract F30602-98-2-0107; to Jon Bredin, Brian Brewington, and Arne Grimstrup for invaluable feedback on early drafts of this paper; and to the anonymous reviewers for their useful and thought-provoking comments.

References

[DR99] Amitava Dutta-Roy. Bringing home the Internet. *IEEE Spectrum*, 36(3):32–38, March 1999.

[Gri99] Corey Grice. When will data change the wireless world? *CNET NEWS.COM*, February 10, 1999.

[LO99] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[MBB+98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. In *Proceedings of the Second International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67, Stuttgart, Germany, September 1998. Springer-Verlag.

[MMBC97] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 1–20, 1997.

[Vig98] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

Foreign Event Handlers to Maintain Information Consistency and System Adequacy *

QUELOZ Pierre-Antoine
Pierre-Antoine.Queloz@bigfoot.com

PELLEGRINI Christian
pell@cui.unige.ch

Centre Universitaire d'Informatique
University of Geneva
Switzerland

April 1, 1999

Introduction

The goal of this paper is to describe novel applications of Mobile Code technology which have not appeared yet but should be feasible with our current knowledge of the domain. These new applications contradict the often-made observation that Mobile Code is just another technique that does not really bring much more possibilities than existing technologies for distributed applications. There is a whole class of problems that have not received much attention yet and that are not well managed by current environments. These are the problems of maintaining consistency of dynamic information and maintaining systems in adequacy with the ever changing requirements of customers.

Our motivation is that, besides the quantitative improvements that most people expect from using Mobile Code, there is also a qualitative benefit which is even more important but not universally recognized now: Mobile Code allows communication with *less conventions* than message passing [5, 3]. Processes interconnected by Mobile Code still have to agree on high level encoding and synchronization primitives but these agreements are only a fraction of what is necessary to communicate. Many context dependent aspects can be encapsulated inside Mobile Code and changed when the context changes. Encapsulation has the same benefits here as in other software engineering domains: it reduces the dependency between components, thus reducing the number of modifications that we must make to software in order to adapt it to new requirements. For this reason we think that it is the best way to cope with systems that are *distributed*, hence not manageable by a single person or organization; that are *dynamic*, because the information they contain must change when the world itself changes; and that are *evolving* since the users discover new ways to use them and the information provided.

The basic idea is to hook foreign Mobile Code to a running application without interfering with its basic functionality and to execute these Mobile event handlers to perform any new task in connection with the application. The interest is that with only a small number of additional constraints in the original design of the system it becomes completely adaptable to future needs and all the information it manages becomes available for new purposes.

*Position paper accepted at the Workshop on Mobile Agents in the Context of Competition and Cooperation — Autonomous Agents Conference — 1999

The problems

In order to build a system that will work for a long period of time we must be able to *anticipate* what is going to happen around it. The first kind of dynamism we have to deal with is the dynamicity of the world itself: people are moving, objects are created, exchanged, destroyed, new books are published, etc. To be useful most systems must take these changes into account and reflect them in the information they contain and offer. We can often anticipate these changes if we build a good model of the world on which our system is based and if we know how to detect events and handle them within our dynamic system.

Broadly speaking most systems that handle events and world dynamicity cope by applying the principle of *publish and subscribe*. This principle of publish and subscribe for event notification is so general that it was already in use long before computers existed. It is the natural way to pass information to interested parties and there is no wonder that it is present in so many different contexts in today's information systems, from electronic mailing lists to "push channels" and inside object-oriented applications underlying the "Observable-Observer" Design Pattern.

It is much more difficult to anticipate how users are going to use a successful information system. If the system has enough users, it is almost certain that they will invent new ways of using it and discover new requirements that were completely unpredictable. To satisfy these new needs, our system must be able to evolve. Experience shows that systems that cannot evolve tend to be obsoleted and replaced by their more powerful competitors.

It is disappointing to see that such a valuable principle as publish and subscribe does not help much to accommodate the potential needs of users. It is a fact that if the designer of a system has not provided the right interface, it will not be possible to access the information needed, even if it is buried somewhere in the system. It will also not be possible to get event notifications, unless some kind of subscription is available in the interface. For this purpose, the designer will have to choose which information is observable and when and how notifications will be propagated to users. With this scheme some extensions are possible, but sooner or later limitations will appear for instance because interesting events are not propagated or because the propagation scheme is difficult to use, or inefficient, etc.

Some widely used systems have even been "victims of their own success": because lots of users with conflicting needs were using them, they could not be further upgraded without "forking" into incompatible products. It is the same phenomenon that occurs with successful protocols like IP, HTTP or HTML: they are installed on millions of hosts, they quickly generate new needs for which new solutions exist but are almost impossible to adopt because of the costs of upgrading every interconnected host and the risk to introduce incompatibilities.

Another concern is the difficulty for people to *collaborate* in order to maintain information systems. Large distributed systems belong to different entities (people, organizations) between which collaboration might not be possible, for instance because there are too many users, or users are too far, or supporting the service is too expensive or all developers are gone. We need to find solutions to keep systems useful without the need for too much collaboration.

Solution based on Mobile Code

To solve the problems described in the previous section, the possibility offered by Mobile Code to encapsulate conventions and adapt protocols to new needs is extremely valuable. So our goal is to use Mobile Code to minimize the number of conventions that would require much anticipation or much collaboration. This solution is not related to a particular technology (Messengers, Mobile Objects, Mobile Agents, strong or weak mobility, etc.) but to the general

Mobile Code paradigm. However, it is not necessarily straightforward to implement with all current Mobile Code environments.

In many cases we think that it is possible to design information systems without giving too much attention to communication issues in a first phase but to have some kind of Mobile Code execution environment integrated within the system itself. The basic way to communicate with such systems is to send Mobile Code for local interactions.

These interactions can only take place between threads triggered by Mobile Code and visible parts of the service (its interface). The interface typically will allow external software entities to retrieve and update information or to watch events occuring in the system. This Mobile Code can be written either by the service designer or by consumers if they have special needs and cannot collaborate with the service designer but have a good understanding of the interface. Encapsulating low level communication in Mobile Code guarantees that the system can remain the same even if the underlying network evolves in unpredictable ways.

Our attempt to design interfaces for distributed services and to describe them with a complex interface definition language [2] has revelead that it is a puzzling and decouraging task; mainly because we have to guess which events will be interesting for service users and what will be the most convenient way to access information. For this reason we think that instead of being hidden behind interfaces, systems should remain open for inspection and allow attachment of Mobile Code. If we remove the outer shell between a system and foreign threads, they will be able to access information more directly. Information itself still needs to be encapsulated at the object level but the granularity is smaller. The consequence is that it becomes possible to work with a much larger set of actions and events.

Mobile Code attached to this kind of system should not disturb its functionality, thus restrictions are necessary. One feasible policy is to let foreign handlers inspect objects through their interface, and install "hooks" that will be triggered when a given object receives a given message (one of its methods is called). Other researchers [4] have reached the same conclusions and identified the need to work at the object granularity and to trigger event handlers when objects are receiving messages.

Examples of applications

One very typical example of lack of information management and bad event handling is the problem of broken links in the World Wide Web. This phenomenon happens all the time because people are referencing or bookmarking interesting pages that do not have a definitive location but can move or be destroyed when site managers reorganize their documents. Mobile Code would be an elegant way to solve this difficult problem: when a user is interested in a page, he writes an agent and sends it to the server. When properly hooked to the server, the agent is ready to be executed when the page is moved or deleted and can do whatever seems appropriate to the user, for instance send an e-mail or fix hyperlinks on the client side.

The systems performing distributed scheduling and distributed resource allocation are also interesting candidates because there are often many dependencies between resources. These dependencies imply frequent constraint checking and complex event handling to keep information up to date. Moreover it is very difficult to foresee which constraints will be imposed by users and business processes on each resource. But it is certain that constraints and dependencies do evolve when people and things move and change. With people having personal electronic organizers and resources such as places or vehicles having their own agenda it is easy to imagine how complex the interrelations can be and how interesting it would be to program agents to check constraints and detect inconsistencies. For instance "my wife and I cannot accept two invitations for the same day" or "there must always be one free seminar

room on Mondays".

We have been very surprised to see how easily anyone who has worked with computers but has no notion of mobility will find realistic examples of this kind, in such diverse domains as multimedia distribution, system administration, source code management, electronic commerce, scientific collaboration, etc. [3]. For this reason we are quite certain that there is a large class of such important problems and not just a few isolated cases.

Technical issues

We should be able to find satisfactory solutions for these problematic applications with our approach of Mobile Code as foreign event handlers. This new approach is certainly not free of practical pitfalls but recent research on Mobile Code has already proposed fairly good answers for most of them:

(1) Memory usage: To limit the consumption of server memory by customers' event handlers we can apply principles of accounting, namely sell server memory to the customer to incite sparing it.

(2) Performance: To keep the main system running, it must not be overloaded by the evaluation of event handlers. We can do it with priority mechanisms, with distribution of the load to more than one server and by keeping the conditions simple (evaluating complex conditions for a lot of event-handlers to decide if they have to be started can quickly become extremely heavy).

(3) Security: Some environments like Java provide authentication and verification of foreign code. Authentication is usually based on encryption and determines whether the code originates from a trusted source and verification is a runtime mechanism to ensure that untrusted code does not perform illegal operations.

There are also some problems that seem more fundamental and challenging with this approach:

(4) Inspection and installation of event-handlers somewhat contradict the principles of encapsulation that we need so much in large systems, they could increase the dependencies between sub-systems and make them more difficult to manage.

(5) We need reciprocal mechanisms to ensure that event handlers do not become obsolete by the actions of users in their own environments.

(6) In a complex system with many interrelated informations and lots of event-handlers there could be circular dependencies leading to a new type of infinite loops. One of the classical solutions is to impose an order or hierarchy on elements to avoid such loops.

(7) An open system with inspection facilities will probably require lots of efforts if secrets have to be hidden, but we can imagine that the system designer hides some classes, encapsulates precious information in objects or uses cryptographic techniques to hide sensitive data.

(8) Most existing systems expose very little of their internal information and events to the outside and are difficult to integrate with the new approach.

Related work

In our recent efforts to build distributed evolvable dynamic information systems we have followed two directions : (1) Using the MØ programming language and Mobile Code environment, we have built a system to manage a library of MØ source code that allows us to edit, store, disseminate and activate five different kinds of source code files. Previously they were difficult to manage because they were all stored in indistinct text files that did not allow to represent their particularities and interdependencies. (2) Maintaining the consistency of documents at

our website and trying to make it accessible to foreign event handlers. This work is at an early stage and has been slowed down by the difficulty to integrate the existing server and HTML editors used at the institute.

There are several other groups working on techniques for large scale event handling and maintaining information coherence [4] but there is only one system of which we are aware that closely approaches our goals of using Mobile Code in distributed information systems: the Stormcast weather information and forecast system [1]. The designers of the service have programmed a set of agents for their customers who can choose parameters and activate agents to monitor the current weather situation and send messages according to their specific needs. This system is interesting because it shows the feasibility of having lots of users executing their agents on the server.

Conclusion

We hope that this paper has convinced the reader that (1) Mobile Code brings flexibility in distributed systems because it has the ability to encapsulate conventions and frees us from the need to pre-install the software that behaves according to these conventions (2) there is an urgent need of flexibility in today's communication protocols and distributed systems because it is impossible to anticipate how a successful system will evolve (3) by letting users add their own event-handlers and interaction protocols to running systems using Mobile Code technology, it could be possible and useful to minimize the needs for collaboration between information providers and consumers (4) there are several interesting problems for which Mobile Code will probably be the best solution because they occur in systems that are distributed thus not manageable by a single person or organization, dynamic because the information they contain must change when the world itself changes and evolving when users discover new ways to use the information provided.

References

[1] JOHANSEN, D., *Mobile Agent Applicability*, in ROTHERMEL, Kurt (ed.), HOHL, Fritz (ed.), MOBILE agents: MA'98: proceedings, Berlin [etc.]: Springer, 1998, Lecture notes in computer science; vol. 1477, ISBN 3-540-64959-X, (1998)

[2] MUHUGUSA, M., *Distributed services in a messenger environment: the case of distributed shared-memory*, Ph.D. Thesis, University of Geneva, 1997.

[3] QUELOZ, P.-A., *Open Directions for Mobile Code*, unpublished slides with presentation notes, 1998, available at http://cui.unige.ch/~queloz/papers/opendir.ps.gz.

[4] ROSENBLUM, D.S., WOLF, A.L., *A Design Framework for Internet-Scale Event Observation and Notification*, Proceedings of the Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the foundations of Software Engineering, Zurich, Switzerland, 1997.

[5] TSCHUDIN, C.F., *On the structuring of computer communications*, Ph.D. Thesis, University of Geneva, 1993.

Tools for Managing Mobile Agent Systems

A Case for Mobile Agent Patterns

Dwight Deugo
School of Computer Science,
Carleton University
1125 Colonel By Drive,
Ottawa, Ontario, Canada, K1S 5B6
deugo@scs.carleton.ca
http://www.scs.carleon.ca/~deugo

Michael Weiss
Business Communications Systems
Mitel Corporation
350 Legget Drive
Ottawa, Ontario, Canada, K1K 1X3
michael_weiss@Mitel.COM

Abstract: In this position paper, we make a case for the development of mobile agent patterns. Patterns have proven extremely useful to the object-oriented programming community. However, of the large amount of pattern research, little effort has been devoted to developing mobile agent patterns. We wish to correct this situation. We believe that the ongoing success of mobile agent systems depends on the development of software engineering principles for them. Patterns are a recognized means to this end, and one that we wish to promote.

Introduction

In most areas of research there comes a time when the researchers begin to understand the principles, facts, fundamental concepts, techniques, architectures, and other research elements in their fields of study. Research into agents, specifically mobile agents, is reaching that time. As evidence of this fact, one need not look any further than the calls for papers to upcoming conferences and workshops on mobile agents. Take for example, the call for papers for the workshop on Mobile Agents in the Context of Competition and Cooperation [MAC3 99]. We find comments such as, "… gaining more widespread acceptance and recognition as a useful abstraction and technology" and, "we are uninterested in papers that describe yet another mobile agent system." The question to answer now is what is important for us to do next as a research community.

To answer that question we must first consider why we do not want to hear about another mobile agent system. We will not answer for the various workshop and program committees, but we can propose one that we hear often from others. Since there are many mobile agent systems and frameworks in existence doing many of the same things, there is no use discussing another one because it too will do the same things and in similar ways. In other words, a new system does not often provide any new insights that are useful to the research community. However, we claim that these 'new' systems do validate, refine and show the reuse of many of the previously proposed and discussed research elements. Moreover, they bring with them additional thoughts, understanding and clarifications of the research elements. The problem is that when reporting on these 'new' systems, these insights get lost in the discussion of the system as a whole, or they are just not reported at all. For example, we have lost track of how many times we have read a paper that indicated it used KQML for the communication between agents and not been able to understand why it was used? There may have been an obvious advantage, or maybe it just did not matter. What we wanted to understand was which forces and context lead to this decision, because if we need to make a similar decision in the future we need this information.

We propose that, since as a research community we have reached a stage where some research elements in mobile agents are well understood and that there are several examples of each, it is time to begin the effort of documenting these elements as software patterns. This is not a matter of documenting the solution and problem surrounding each research element; this material is already evident in most agent papers. We need to go further and deeper in order to understand the forces and context of the problems that give rise to the proposed solutions. These are the undocumented and often misunderstood features of the research elements, which need to be elaborated before agent systems can enter the mainstream of software engineering and business applications.

Since many are not familiar with software patterns, and those that are think of them as only problem and solution pairs, we introduce patterns and pattern languages in order to help with their understanding. Next, we enhance our argument as to why agent patterns are important for agent research. We then compare and contrast agent patterns with their object-oriented counterparts. Finally, we present the current layout of the agent pattern landscape, identifying what some have started to do and what else we feel needs to be done.

What are Patterns and Pattern Languages?

Software patterns have their roots in Christopher Alexander's work [Alexander 79] in the field of building architecture. After reading his work, it was clear to software engineers that, like building designs, there are many

recurring problems and solutions used in the design of software systems. Unfortunately, they noted that many of these combinations were hard to find except for in the minds of the most experienced developers, for if they were, projects would have been built on time, within budget and without bugs! Moreover, knowing the problem and solution were not enough. Software engineers needed to know when the solutions were appropriate for the given problems.

Alexander's proposed the following definition for a pattern:

- A three-part rule which expresses a relation between a certain context, a problem, and a solution.
- As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
- As an element of language, a pattern is an instruction, which shows how the spatial configuration can be used, repeatedly, to resolve the given system of forces, wherever the context makes it relevant.
- The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must created it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate the thing

Although there are different pattern formats, the minimal format contains the following headings or ones dealing with similar subject matters.

- **Name:** As the saying goes in the object-oriented community, a good variable name is worth a thousand words and a good pattern name, although just a short phrase, should contain more information than just the number of words would initially suggest. Would the word agent be a good pattern name? The answer is no. Although it means a lot more than the single word suggests, it has too many meanings! One should strive for a short phrase that still says it all.
- **Problem:** A precise statement of the problem to be solved. Think of the perspective of a software engineer asking himself, How do I … .? A good problem for a pattern is one that software engineers will ask themselves often.
- **Context:** A description of a situation when the pattern might apply. However, in itself the context does not provide the only determining factor as to situations in which the pattern should be applied. Every pattern will have a number of forces that need to be balanced before applying the pattern. The context helps one determine the impact of the forces.
- **Forces:** A description of an item that influences the decision as to when to apply the pattern in a context. Forces can be thought of as items that push or pull the problem towards different solutions or that indicate trade-offs that might be made [Coplien 96].
- **Solution:** A description of the solution in the context that balances the forces.

Other sections such as resulting context, rationale, known uses, related patterns and implementation with sample code are usually included to help with the pattern's description.

A good pattern provides more than just the details of these sections; it should also be generative. Patterns are not solutions; rather patterns generate solutions. You take your 'design problem' and look for a pattern to apply in order to create the solution. The greater the potential for application of a pattern, the more generative it is. Although very specific patterns are of use, a truly great pattern has many applications. For this to happen, pattern writers spend considerable time and effort attempting to understand all aspects of their patterns and the relationships between those aspects. This generative quality is so difficult to describe that Alexander calls it "the quality without a name", but you will know a pattern that has it once you read it. It is often a matter of simplicity in the face of complexity.

Although useful at solving specific design problems, we can enhance the generative quality of patterns by assembling related ones, positioning them among one another, to form a pattern language, enabling us to use them to build systems. For example, individual patterns might help you design a specific aspect of your mobile agent, such as how it models beliefs, but a pattern language might be able to help you build all types of mobile reactive agents.

Agent pattern languages are very important for agent patterns to be successful. Forcing each pattern to identify its position within the space of existing patterns is not only good practice, it is also good research. In other words, all agent patterns should be part of an agent pattern language. It is not only helpful to you, but to all those other software engineers who will use the patterns to develop their systems in the future.

Why is Research into Patterns Important for Agent Research?

For any software system to be successful and run safely, it must be constructed using sound software engineering principles, and not constructed in an add-hoc fashion. Unfortunately, much of agent development to date has been done ad hoc [Bradshaw, et al., 1997], creating many problems – the first three noted by [Kendall et, al, 1998]:

1. Lack of agreed definition of an agent
2. Duplicated effort
3. Inability to satisfy industrial strength requirements
4. Difficulty identifying and specifying common abstractions above the level of single agents
5. Lack of common vocabulary
6. Complexity
7. Only goals and solutions presented

These problems limit the extent to which "industrial applications" can be built using agent technology, as the building blocks have yet to be exposed. Objects and their associated patterns have provided an important shift in the way we develop applications today, since the level of abstraction that we develop at is greater than doing functional and imperative programming. Since we believe that agents are the next major software abstraction, we find it essential to begin an effort to document that abstraction so that others can share in the vision. Patterns provide a good means of documenting these building blocks in a format already accepted by the software engineering community. Patterns also have the added benefit that no unusual skills, language features, or other tricks are needed to benefit from them [Lange and Oshima, 1998].

Are Agent Patterns different from their Object-Oriented Design Patterns?

Since many mobile agent frameworks, such as Grasshopper, AgentSpaces, Voyager, and Aglets, are implemented using Java, an object-oriented language, we argue that there must have been a few object-oriented design patterns used by the developers constructing them. Some patterns such as Mediator, Adapter, Broker, Strategy, and Composite, have been explicitly identified as being useful for the development of agent systems. [Kendall et al., 97]. However, since agents and agent systems are more dynamic and able to adapt to their environments than object-oriented systems, there must obviously be other patterns that are applicable only to them. We have already seen some of these patterns documented [Lange and Oshima, 98, Silva and Delgado 99], but we are positive there are a great many more.

In general, we find the format and structure of Agent Patterns similar to Object-Oriented Design Patterns. We also find that since many mobile agent frameworks are implemented using object-oriented technology, most object-oriented design patterns are applicable to them. However, differences in the way agents communicate, their level of autonomy and intelligence, and the fact that they are often highly mobile, has created a void not filled by existing patterns: agent, object, or otherwise.

One criticism of the previous breed of pattern writers is that they have not done a good job of relating their patterns to others – a pattern language. This is starting to change now as pattern languages become more frequent, but historically patterns have typically only linked themselves to others without identifying exactly what the relationship is. This is a problem that agent patterns writers must consider immediately. Describe how your patterns relate to others and how they can be used together or separately. Create a pattern language that can be used to build things: mobile agents and mobile agent systems. Do not create patterns that will only frustrate their readers, forcing them to discover how to apply the patterns together.

The Agent Pattern Landscape

The application of design patterns has shown us that it is very important and useful to start formalizing the experiences of developers and relating these formalizations with one another. There are a small number of agent patterns compared to the number object-oriented design patterns. As a consequence little work has been devoted to classifying them and, as mentioned in the last section, to defining pattern languages for them. The few classifications already proposed include the following:

- **Agent Patterns:** deal with the architectures of agents and agent-based applications [Kendall et al. 98, 97; Silva and Delgado 98; Aridor and Lange, 98].
- **Communication Patterns:** deal with the way agents communicate with one another [Deugo and Weiss, 99].

- **Travelling Patterns:** deal with various aspects of managing the movement of agents such as routing and quality of service Lange and Oshima 98].
- **Task Patterns:** deal with the breakdown of tasks and how these tasks are delegated to one or more agents Lange and Oshima 98].
- **Interaction Patterns:** deal with the way agents locate one another and facilitate their interactions [Lange and Oshima 98].
- **Coordination Patterns:** deal with managing dependencies between agent activities. [Tolksdorf 98]

As a general comment, patterns cover many different levels of abstraction. For example, some of them are used to describe the structure of an mobile agent system. Other patterns support the structure of agents and their relationships with different agents. While other patterns can be used to specify the design aspects of individual agents. The important feature here is not in developing the definitive classification. Rather it is more important for the mobile agent community to identify, specify and agree on the abstractions so that we can provide a common vocabulary for discussing and enhancing them, and, more importantly, building industrial strength mobile agent applications based on well-grounded software engineering principles.

Conclusion

What is in the future of mobile agent pattern research? Our prioritized list is as follows:

- **Identify an initial set of agent pattern classifications:** These classifications are to help focus pattern writers on targets that are of the greatest importance to those developing 'real' mobile agent systems.
- **Identify pattern languages within each classification:** These pattern languages are for pattern writers to develop and extend and will permit writers to position their new patterns within a know space of existing patterns.
- **Write the patterns.**

It is at MAC3 that we wish to debate and discuss these foundations with the participants in order to fill in the details of how to proceed and why bother at all. We feel it necessary to remind those involved with mobile agent research to not only write about solutions. Think, discuss, and write about the problems their solutions are intended to address and what context and forces led them to that particular solution. In short, we believe that following this approach, we will not have to read about "yet another mobile agent framework" anymore. Rather, we will be able to read and understand what problems a mobile agent system solves, and when we **should** consider using the approach!

References

Aridor, Y., Lange, D., "Agent Design Patterns: Elements of Agent Application Design", Proceedings of the Second International Conference on Autonomous Agents (Agents 98), ACM Press, 1998, 10-115.

Bradshaw, J.M., S. Dutfield, P. Benoit, J.D. Woolley, "KaoS: Towards and Industrial-Strength Open Distributed Agent Architecture", J.M. Bradshaw (Ed.), Software Agents, AAAI/MIT Press, 1997.

Coplien, J.O., "Software Patterns", SIGS Management Briefings Series, SIGS Books & Multimedia, 1996.

Deugo, D.L, "Communication as a Means to Differentiate Objects, Components and Agents", submitted to TOOLS USA 99, 1999.

Kendall E. A., P.V. Murali Krishna, Chirag V. Pathak, C.B. Suresh, "Patterns of Intelligent and Mobile Agents", Autonomous Agents '98 (Agents '98), 1998

Kendall E. A., M.T. Malkoun and C.H. Jiang , "Multiagent System Design Based on Object Oriented Patterns", The Report on Object Oriented Analysis and Design in conjunction with The Journal of Object Oriented Programming, June 1997

Lange, D.B., M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", Addison Wesley, 1998.

MAC3, "Mobile Agents in the Context of Competition and Cooperation", Autonomous Agents '99 , http://mobility.lboro.ac.uk/MAC3, 1999.

Silva A. and J Delgado, "The Agent Patterns: A Perspective from the Mobile Agent System Point of View", EuroPLoP '98, 1998.

Tolksdorf, R., "Coordination Patterns of Mobile Information Agents", Proceedings of Cooperative Information Agents II, Second International Workshop, CIA'98, Springer, 1998, 246-261.

Mobile Agent Applications

Mobile agents in an electronic auction house*

Qianbo Huai and Tuomas Sandholm
{qh2, sandholm}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130-4899

Abstract

This paper presents *Nomad*, a mobile agent system for electronic auctions. It has been integrated with *eAuctionHouse*, our next generation Internet auction server. To our knowledge, *eAuctionHouse* is the first Internet auction site that supports combinatorial auctions, bidding via graphically drawn price-quantity graphs, and participation of mobile agents. It supports mobile agents so that a user can have her agent actively participating in the auction(s) while she is disconnected. Mobile agents that execute on the agent dock which is on (or near) the host machine of the auction server also reduce the network latency—a key advantage in time-critical bidding. Our auction server uses the Concordia agent dock to provide mobile agents a safe execution platform from where they can monitor the auctions, bid, set up auctions, move to other hosts, etc. The user has the full flexibility of Java programming at her disposal when designing her agent. We also provide an HTML interface for non-programmers where the user can specify what she wants her agent to do, and the system automatically generates the Java code for the corresponding mobile agent, and launches it. Some of these predesigned agents are alerting tools, others bid optimally on the user's behalf based on game theoretic analyses. This helps put novice bidders on an equal footing with experts. Finally, we discuss automated coalition formation among bidder agents.

1 Introduction

As the Internet gradually moves into mainstream culture, electronic commerce is becoming an important mechanism for conducting business. It helps merchants and consumers reduce business costs and enables customized delivery of goods and services. Electronic auctions are emerging as one of the most successful ecommerce technologies.

Several successful commercial Internet auction sites exist - such as eBay[1] and Onsale[2] - and academic auction houses have recently appeared on the Internet [4, 9]. Our motivation in developing an auction server, *eAuctionHouse*,

*This material is based upon work supported by the National Science Foundation under CAREER Award IRI-9703122, Grant IRI-9610122, and Grant IIS-9800994.

[1]http://www.ebay.com

[2]http://www.onsale.com

is to prototype novel next generation features, and test their feasibility, both computationally and in terms of consumer ease of use. To our knowledge, *eAuctionHouse* is the first, and currently only, Internet auction site that supports combinatorial auctions [3, 6, 7, 5], bidding via graphically drawn price-quantity graphs [8], and mobile agents. This paper focuses on the mobile agent component.

2 *eAuctionHouse*, a next generation electronic auction house

eAuctionHouse is a free-to-use Internet auction prototype. It allows users across the Internet to buy and sell goods as well as to set up markets. It acts as a third party site, so both buyers and sellers can trust that it executes the auction protocols as stated. It offers a wide range of auction types each of which is determined by specifying a combination of auction parameters. Both auctions and bids are specified by a set of parameters. In a single-sided auction, there is only one seller (buyer) selling (buying). In a double auction, both buyers and sellers can submit bids (offers to buy) and asks for (offers to sell) goods. An auction setting is determined by whether it is a single or double auction, whether there is one or multiple items, and whether there is one or multiple units of each item. Auction settings in our server further differ based on whether or not bids in multi-unit auctions can be partially filled or whether complete matches are required. For different types of auctions, different types of bids can be accepted, bidders may use different bidding strategies, and different types of pricing schemes may be applied to determine the winning price(s). Due to space limitation, combinatorial auctions and quantity-price graph bidding are not discussed here. The reader can find in [8] a more detailed discussion regarding all the auction settings supported by *eAuctionHouse* and the relationships among auction settings, bid types, and pricing schemes.

To create auctions or to submit bids, a user can visit the world-wide-web page of *eAuctionHouse*, send a formatted text string directly through a TCP/IP network connection, or use mobile agents.

3 Why mobile agents?

There are two classes of advantages of mobile agents in electronic auctions: those stemming from the use of agents, and those from mobility.

The benefits of using agents include the following.

- An agent can monitor the auction events that the user has deemed relevant. When such events occur, the agent can alert the user. This avoids

the need for the user to keep polling the auction repeatedly.

- Compared with parameter specified bidding, bidding agents provide the user with more flexibility when customizing her own bidding strategy.
- An agent can make decisions based on all available information that the bidder considers relevant.
- Prototypical bidding agents can be analyzed game theoretically off-line so that they will bid optimally on the user's behalf in given auction settings. This puts expert bidders and laymen on a more equal footing for ecommerce.
- Agents can be built to track bids in multiple auction houses, looking for the best deal and/or coordinating the user's multiple bids, e.g. submitting bids to multiple sites for buying the same good but at anytime allowing at most one bid in the winning position.

The benefits from mobility include the following.

- Mobile agents are not very sensitive to the quality of network communication.
- A user's computer only needs to be connected to the network occasionally. A user can connect, launch an agent, and disconnect. This avoids the need of being connected all the time.
- If the information transferred between the agent and the auction exceeds the code size of the agent, mobility reduces bandwidth usage because the agent communicates locally at its destination rather than having to send data back and forth. This reduces network traffic and latency by reducing the amount of data transferred around the network.
- Mobile agents can potentially take advantage of the available services distributed across the network, e.g. travel to and execute on powerful servers with excess CPU time and disk space. This can be pertinent for bidder agents for example if their bidding strategies include complex computations such as statistical analysis, projection, etc.
- Mobile agents potentially allow users to extend the functionalities offered by servers, i.e. they execute as an integral part of the server.

In the context of an electronic auction, the use of mobile agents frees the user from having to monitor the auction constantly. It also can reduce network latency, a feature which is of key importance in time-critical bidding, such as making decisions just before an auction will close. A mobile agent traveling

to a server not only has the advantages of executing the user's own bidding algorithm but also does not have to transfer over the network the auction information that the agent uses for determining its bidding actions.

Here we give an example of a mobile bidding agent. Let us imagine that a user with access to the *eAuctionHouse* is at some remote location. She finds an auction of a new notebook computer with an ascending open-cry first-price auction protocol. Say that in this particular auction setting, the highest bid is shown when placed. Bids are accepted by the auction until no user is willing to increase the price anymore. At that time the notebook is sold to the bidder who offered the highest price and the winner is charged the price of her bid. Say that the auction is at a remote location, so the communication links are problematic. The auction may be open for hours but the user is not able to monitor and adjust the bidding price constantly. A mobile agent can help in this situation. The user can give it a reservation price and launch it. The agent moves to the server site and stays there monitoring the current highest price. It bids a small increment more than the current highest price, and stops if its reservation price is reached. When the auction closes, the agent reports the result to the user.

Both Onsale and the Michigan Internet AuctionBot [9] have solutions other than mobile agents to help the user in the above scenario. Onsale has Bid Maker which allows the user to enter the maximum price she is willing to pay. eBay has a similar proxy bidder "agent". As long as the auction is open and the maximum price has not been reached, Bid Maker bids at the minimum price necessary to make the user a winner. Bid Maker limits the user's choices of her bidding strategies. For example, when a user's valuation of the good depends on others' bids, such a simple agent is no longer optimal, but the agent should update its valuation dynamically based on the others' bids so far (taking into account the effect of the winner's curse [1]). Bid Maker is also unable to support bid coordinating across multiple auction houses.

The Michigan Internet AuctionBot might be viewed as supporting agents in the sense that it provides a TCP/IP level message protocol by which agents can participate in the auction. Mobile agents are not supported. Without mobility, this solution is sensitive to the quality of network communication and increases the network traffic by frequently transferring auction information.

4 *Nomad*, a mobile agent system

Nomad is the mobile agent system integrated with *eAuctionHouse*. It allows mobile agents to travel to the site where the *eAuctionHouse* resides. Mobile agents actively participate in the auctions on the user's behalf even when the user is disconnected from the network. Network traffic and latency are

reduced, and the agents have shorter response time to changes in the auction than users might. The amount of time for computing bidding strategies may also be lowered when agents execute on a powerful server. Mobile agents need not necessarily be bidding agents. They may be information collection or price distribution learning agents as well as agents for setting up auctions. When multiple distributed *eAuctionHouses* are installed across the network, multiple *Nomads* help to form a virtual electronic auction site network, which allows mobile agents to travel through the network to find deals. Also implemented in *Nomad* is a mobile agent control scheme. After registering itself to the server, a mobile agent can be seen and managed in its creator's user portfolio. For example, the user can kill the agent when she no longer wants the agent to execute.

The high-level architecture of a *Nomad* system consists of three main components: an agent dock, an agent manager, and a database for keeping information of the mobile agents. The agent dock is the place where mobile agents reside and execute. As the basis of our agent dock we use the Concordia[3] system from Mitsubishi Electric Information Technology Center America. Concordia is a full-featured framework for the development and management of network-efficient mobile agent applications. Concordia itself is a Java application and supports mobile agents written in Java. Application interfaces are provided in Concordia for sending agents around the network. Concordia agents process data at the data source. Network transport is hidden from applications, developers and users. Typically, a Concordia agent has an itinerary which can be seen as a list of network addresses where the agent desires to go. Associated with each address is an action, a Java class method, which is to be executed when the agent travels to the associated site. The itinerary may be altered dynamically during the agent's trip. Agents can also collaborate with the help of an event distribution mechanism and other services.

The agent manager helps users control their own mobile agents through *eAuctionHouse*. Agents can be deleted from *eAuctionHouse* by their owners. An event distribution framework is used by the agent manager to notify agents when the auction information they are interested in is altered.

After traveling to the site of *Nomad* and being docked on the Concordia system, an agent connects to the agent manager and reports its information which is logged into a database. Later on, the agent's information can be retrieved using the application interface. Actions requested by the user, such as deleting a particular agent, are forwarded from the *eAuctionHouse* to the agent manager which then distributes an agent kill event to all the docking agents. After receiving an event addressed to itself, the agent performs the appropriate actions.

[3] http://www.meitca.com/HSL/Projects/Concordia/

5 Automatic generation of mobile agents

Users are provided with the option of programming their own mobile agents in Java. This allows maximum flexibility in what agents can do. However, in order to speed up agent generation and to enable non-programmers to create agents, there is also a system for automatic generation of agents based on filling out HTML forms. The difference between *Nomad* and Onsale's Bid Maker is that our agents are mobile and HTML forms are not the only means to create them. The following types of parameterizable mobile agents are currently available for automatic generation.

1. The *information agent* monitors an auction and sends email to the user when specified events occur. Using this agent, the user does not have to poll the auction, but gets notification of important events immediately.

2. The *incrementor agent* implements the dominant strategy on the user's behalf in single-item single-unit ascending open-cry first-price private value auctions. It bids a small increment more than the current highest price, and stops if the user's reservation price is reached. With this agent, the user does not have to follow the auction, and her dominant strategy in these settings is to report her valuation truthfully to the agent.

3. The *N-agent* underbids optimally on the user's behalf in single-item single-unit sealed-bid first-price auctions where the number of bidders, N, is known, and the bidders' private valuations are independently drawn from a uniform distribution. Specifically, the symmetric Nash equilibrium strategy is to bid the user's valuation times $\frac{N-1}{N}$ [2]. The user is then motivated to reveal her true valuation to the agent.

4. The *control agent* goes to an auction and submits very low noncompetitive bids. It is a speculator's tool to artificially increase the number, N, of bidders in an auction to mislead others, e.g. the *N-agent*. For example, it is in the seller's interest to submit control agents so that *N-agents* would bid higher.

5. The *discover agent* computes the expected gain from bidding a small amount more than the current highest price according to the agent's current distribution of the user's valuation. This is intended for settings where the user does not know her exact valuation for the item, instead has a probability distribution over it. In the future, the probability distribution could be updated by new events, or by what others have bid in non-private value auctions.

Figure 1 shows the first step of creating a mobile agent without programming. In this example, a user, Alice, is going to create an agent to bid in an

auction. For the given auction type, the system recommends three of the five agent types. Therefore, the radio buttons of the other two agent choices are not clickable for this particular auction type. Alice is interested in creating an *N-agent*.

Figure 1: *Step 1 of agent creation: choosing an agent type from a set of prototype agents.*

Figure 2 shows the second step for creating an *N-agent*. Alice specifies the parameters: user identification number, password, email address used by the mobile agent for reporting, agent name for agent management, and Alice's valuation (reservation price) of the good being auctioned. When Alice clicks the create button, the automatically generated mobile Java agent takes these parameters, travels to the agent dock, ecommerce.cs.wustl.edu, docks there, and bids at the *eAuctionHouse*.

Figure 2: *Step 2 of agent creation: setting the parameters for the agent.*

6 Support for automated coalition formation among bidder agents

In this section we present some thoughts on potentially using mobile agents for automated coalition formation. This is part of our future research.

Economic efficiency can sometimes be improved if bidders form coalitions. Consider an auction in which a seller is selling one good. One buyer wants part of the good and another one requires the remaining part. The sum of the price that they are willing to pay exceeds the highest price offered for the whole good from other bidders. By forming a coalition, both the two buyers and the seller are better off. However it is difficult for users across the Internet to form coalitions while bidding online. There are two main barriers. First, finding partners can be time consuming. Second, bidders do not necessarily trust each other without a binding contract. Issues arising in coalition formation include

who is in charge of bidding, what happens if some bidders refuse to pay after the coalition's bid wins, how much each participant has to pay if the coalition wins, etc.

To support automated coalition formation, we propose the possible use of mobile agents. Finding potential coalition partners may not be that difficult. With an appropriate communication mechanism, it is easier for an agent to locate potential partners than a person sitting at a computer. The agent may search in a public place where bidders or agents put partial bids for the hope of being combined by others, for example. Agents usually search orders of magnitude faster than humans. Furthermore, agents' time is not as costly.

To solve the trust problem, a third party site might be necessary to help mobile agents form coalitions. At the third party site agents could sign binding contracts, and check the agents' owners' credit history and reputation.

Collusion can improve the economic efficiency as discussed above. However, it involves speculation costs and sometimes causes revenue loss for the auctioneer. For example, bidders in an auction can coordinate to keep their bids artificially low so as to get the item at a lower price than they otherwise would. Considering the user numbers and diversity in most Internet auctions, it is highly unlikely that bidders across the Internet can establish a single coalition for an auction so that bids stay artificially low. Therefore it may be the case that automated coalition formation contributes to the positive side more than to the negative side in the Internet auction setting.

7 Conclusions

This paper presented *Nomad*, a mobile agent system for electronic auctions. It has been integrated with *eAuctionHouse*, our next generation Internet auction server. The reader is invited to visit the site and test the agents (`http://ecommerce.cs.wustl.edu`).

Our auction house supports mobile agents so that a user can have her agent actively participating in the auction(s) while she is disconnected. Mobile agents that execute on the agent dock which is on (or near) the host machine of the auction server also reduce the network latency—a key advantage in time-critical bidding. Our auction server uses the Concordia agent dock to provide mobile agents a safe execution platform from where they can monitor the auctions, bid, set up auctions, move to other hosts, etc. The user has the full flexibility of Java programming at her disposal when designing her agent. We also provide an HTML interface for non-programmers where the user can specify what she wants her agent to do, and the system automatically generates the Java code for the corresponding mobile agent, and launches it. Some of these predesigned agents are alerting tools, others bid optimally on

the user's behalf based on game theoretic analyses. This helps put novice bidders on an equal footing with experts.

Future research includes developing additional prototype agents based on new game theoretic analyses. Also, automated coalition formation by bidder agents brings new challenging problems which will be further studied in the continuing development of *eAuctionHouse* and *Nomad*.

References

[1] P. Milgrom. Auctions and bidding: A primer. *Journal of Economic Perspectives*, 3(3):3–22, 1989.

[2] E. Rasmusen. *Games and Information*. Basil Blackwell, 1989.

[3] S. J. Rassenti, V. L. Smith, and R. L. Bulfin. A combinatorial auction mechanism for airport time slot allocation. *Bell J. of Economics*, 13:402–417, 1982.

[4] J. A. Rodriguez-Aguilar, P. Noriega, C. Sierra, and J. Padget. FM96.5: A Java-based electronic auction house. In *In Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'97)*, 1997.

[5] M. H. Rothkopf, A. Pekeč, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.

[6] T. W. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 256–262, Washington, D.C., July 1993.

[7] T. W. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, 1999. Extended version: Washington University, Department of Computer Science technical report WUCS-99-01.

[8] T. W. Sandholm. *eMediator*: A next generation electronic commerce server. Technical Report WUCS-99-02, Washington University, Department of Computer Science, 1999.

[9] P. R. Wurman, M. P. Wellman, and W. E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the Second International Conference on Autonomous Agents (AGENTS)*, pages 301–308, Minneapolis/St. Paul, MN, May 1998.

A Partitioning Model for Applications in Mobile Environments

Alexander Schill[1], Albert Held[2], Thomas Ziegert[1] and Thomas Springer[1]

[1] Dresden University of Technology, Department of Computer Science, Institute of Operation Systems, Data Bases and Computer Networks, Mommsenstr. 13, D-01062 Dresden, Germany
{schill, ziegert, springet}@ibdr.inf.tu-dresden.de
[2] DaimlerChrysler, Research and Technology 3, Wilhelm-Runge-Str. 11, D89081 Ulm, Germany albert.a.held@daimlerchrysler.com

Abstract: Today mobile devices are an integral part of the execution environment of many distributed applications. The new application domain of mobile computing brought up by this fact introduces problems special to this area. Most of the currently available applications can't handle mobility and frequently changing network media. In this paper a partitioning model is introduced which combines various techniques used to solve the special problems of mobile computing and to facilitate the adaptation of application behavior according to the execution environment. Main points are the proxy approach and the use of mobile agents as application components. The model focuses on a pair of generic proxy agents which are placed at both sides of a wireless connection to control the data transfer between a mobile host and the wired network.

1 Introduction and Motivation

Today mobile devices are an integrated part of the execution environment of many distributed applications. This fact has brought up a new application domain called mobile computing which offers the information access anywhere and anytime but also introduces new problems special to this application area (e.g. frequent disconnections and low bandwidth). Most of the currently available applications can't handle mobility and frequently changing network media. They assume a static environment and often high bandwidth connections typical in LANs. Because of the increasing use of mobile devices and the variety of their hardware resources, it is desirable that applications react and adapt to the frequently changing execution environment. For instance the transfer and the visualization of web pages should be adapted to the available bandwidth to the receiving host and the hardware resources on the mobile device.

Several Solutions have been published introducing environment aware, adaptable applications. Disconnected operations [1,2] are used to handle frequent disconnections. In combination with prefetching and caching strategies, work can be continued without a network connection. Other approaches use data reduction techniques to reduce the data volume, which has to be transferred over wireless links. Examples are the filtration and compression of data according to its type [3]. To implement such techniques an intermediate component called proxy is used. This approach offers the possibility to improve even legacy software, especially client/server applications, without or with very little changes of the existing components.

The paradigm of mobile agents offers new solutions for the problems named above. An agent can be sent to the wired network and during the execution of the agent no connection to mobile host is required. The mobile agent acts autonomously and sends results back when a new connection to its originating host is established.

In this paper a partitioning model is introduced which combines various techniques used to solve the special problems of mobile computing and to facilitate the adaptation of application behavior according to the execution environment. Main points are the proxy approach and the use of mobile agents as application components. The model focuses on a pair of generic proxy agents which are placed at both sides of a wireless connection to control the data transfer. Some major ideas of the model can be found in [10].

Section 2 contains some fundamental considerations and work related to our approach. In section 3 the generic partitioning model is described. In section 4 we present some performance results measured using a sample application. A conclusion is drawn in section 5.

2 Foundations and Related Work

Frequent disconnections are one problem of wireless communication media. There are two ways to handle this issue. First there are *disconnected operations* enabling the user of a mobile device to continue his work even when the connection to the network is lost. The file system CODA [2] realizes this technique by caching files locally in the connected state and accessing only local data if no connection to the network is available. The second possibility is offered by mobile agents. They can be sent to hosts placed on the wired network side and are able to continue their work even if the connection to the mobile host is closed. Results of such an *autonomous operation*[1] can be delivered after a new connection to the mobile device is established.

Caching can reduce the network load by keeping frequently used data locally. By applying a *prefetching* strategy high delay times can be kept hidden to the user. To work efficiently this strategy needs to observe user actions to decide what data has to be prefetched. In the case of wrong decisions bandwidth is wasted but this technique can be used to exploit unused bandwidth. Further improvement can be reached using *lazy write-back*. Data is only written back to a server in the wired network after a modification in the cache and after the access is ended (e.g. the close operation for a file has been called). These techniques need additional resources on the mobile device. Restoring the consistency between cache and source data in the network when switching from disconnected to connected state needs also additional effort (see [1]).

The frequently changing bandwidth and the low available bandwidth in general can be addressed by data reduction prior to the transfer. The deployment of *compression algorithms* (e.g. lzh) reduces the volume of all kinds of data. *Lossy compression* (e.g. JPEG and MPEG) or conversion according to the type of data can be used to reduce the volume of multimedia data. For example, the size or the color depth of a picture could be changed.

Some techniques can decrease the network load by dropping data less important for the overall information of structured data. Therefore additional information at the application level is needed to assess the importance of the parts within a data structure. Hierarchical data (like HTML pages) is well suited to apply *filtering methods*. The importance of data is included in the hierarchy. The higher the position in the hierarchy the more important is the data for the overall information of the structured data. Dropping of data depends on the available bandwidth. The lower the bandwidth the fewer levels of the hierarchy are transferred. Data *outlines* can be created from structured data which decreases the amount of data to transfer. For instance a HTML page could be reduced to headlines and links with placeholders for the other data. The full information only has to be transferred when a section or a placeholder is selected for viewing by the user. *Lazy evaluation* can also reduce the network load. Only actually used data is transferred (the part of a HTML page which is currently visible) (see also [11]).

Disconnections can be handled generally by applying *queuing*. When a connection breaks enqueued data can be transferred via a new established connection. In the same manner the data transfer can be delayed if the bandwidth of the currently available connection is not sufficient. Queuing will not reduce the network load but it enhances the behavior of applications during disconnections.

---

[1] In the literature the phrase disconnected operation is sometimes used for both of the described operations. We use the expressions “disconnected” and “autonomous” to distinguish between both types.

A general approach to support mobile environments is the partitioning of applications and distribution of the components between the mobile device and hosts in the wired network. The aim is the dynamic adaptation of applications in reaction to changes within their execution environment (e.g. quality of connection to the mobile device and availability of resources).

An often used method is the introduction of an intermediate component placed between client and server components. This so called proxy senses its execution environment and adapts the communication over the link to the mobile device according to the available communication media.

An adaptation of the data stream can be done in the network infrastructure [6]. In this approach the amount of data is reduced using data type specific lossy compression and conversion. This functionality is separated from the client and server components and is part of an intermediate proxy within the network infrastructure.

In [5] the main task of the intermediary is filtering the data stream to the mobile device. Therefore the following methods are applicable: optimized protocols, selective dropping of structured data, compression or a deferred data transfer. By using a proxy component, client and server components can remain unchanged or can be used with very little changes.

The approach of partitioning an application is also used in [3]. Data and functionality of an application is partitioned into hyperobjects which are linked hierarchical objects managed by the system. These hyperobjects contain base types such as text, graphics, untyped data and functions. Based on this structure caching, prefetching and data reduction are used to exploit the wireless links efficiently. Data reduction is done by selecting a subset of structured data for the transfer according to the available network resources, application specified data priority and filters.

All the techniques described above can be used to adapt application behavior to the parameters of the available network connection and execution environment. Filtering, compression and conversion of data, outlines and lazy evaluation reduce the amount of data which has to be transferred. These techniques can be implemented as an application component placed at a stationary host. Code for caching, prefetching and lazy write-back has to be placed on the mobile device. Beside these restrictions not all techniques can be combined. While lazy evaluation only transfers data really needed, prefetching transfers data that will possibly never be needed. A partitioning model which combines some of the described techniques is introduced in the next section.

3 The Partitioning Model

3.1 A First Approach

In [4] a traffic telematics application was presented which was based on mobile agents as application components. The application has been partitioned into several parts which were distributed between a mobile device and hosts in the wired network. An agent acts as proxy for the mobile device within the wired network. Information is processed and filtered before it is transferred to the mobile device. The proxy agent is assigned only to this application. A generic partitioning model was also introduced, where a single proxy agent acts for several applications at the mobile host.

The application partitioning model we introduce in this paper is a refinement of this general approach. It is based on the experiences obtained from the traffic telematics application. In general each application needs a component at the mobile device. This component contains the user interface, functions to interact with the user and the components in the network (e.g. configuration, communication and control) and functions independent from the network.

Additional components can be placed at the wired part of the network. These components contain functions related to resources on the wired network side and adapt the data stream according to the communication link to the mobile device. The component in the wired network can be divided to additional subcomponents to facilitate parallel execution of tasks and local communication by migrating these subcomponents to the host of the communication partner.

By implementing this first model in a straightforward way (like the telematics application), it provides disconnected operations by using caching and prefetching and autonomous operations due to the use of mobile components, reduces the network load by filtering and handles disconnections by queuing. But the intermediate component per application which facilitates functions that are generic, could be used by more than one application. Therefore we have further refined our model to separate generic and application related functions of the proxy.

3.2 A Generic Model

The application partitioning model is based on two pairs of components which contain generic and application related functions of the proxy. The generic component placed within the wired network acts as a permanent representation of the mobile device. All communication attempts of the application involving the wireless link are mediated by the generic components.

There are some restrictions with respect to the placement of code which implements the techniques described in section 2. *Caching, prefetching* and *lazy write-back* are realized by the application related component on the mobile device because these techniques need information only available at application level to work effectively. *Caching* and *lazy write-back* can also be applied in a more generic fashion and therefore be realized by the generic component. *Compression* and *conversion* depend on the data type which is generic. *Queuing* improves the disconnection handling of all applications in common. Data can be filtered using application related information. Therefore filtering is done by the application related component within the wired network. Packets can also be filtered using their routing information (e.g. source and target and priority) which is common for all applications. *Lazy evaluation* and *outlines* depend on information available at application level.

In consideration of the above described restrictions the partitioning model was designed as shown in figure 1. It focuses on a pair of generic components which are placed at both sides of the link to the mobile device. Furthermore there are components placed on the mobile device and within the wired part of the network.

Our partitioning model was designed according to the following principles: Information should be used where it is available. Low bandwidth, high latency communication media require an adaptation (volume reduction, adaptation of the send and receive behavior), which can be done in an generic and application specific way, so the system should incorporate both possibilities. Application components should be, as much as possible, dispersed within the wired network infrastructure. For instance, the application can be partitioned according to the structure of the used servers.

Furthermore our model decouples application components from the movement of the mobile device.

3.2.1 The Local Application

This component may be for instance a client of an existing client/server application. In newly developed applications, only one component on the mobile device will exist (see next section).

3.2.2 The Application Related Component on the Mobile Device

This component contains the user interface, functions to process local data (e.g. test of the consistency of user data, configuration, and control of the other components) and application related functions of the proxy as described above. In this component user actions can be observed and application related information can be used to effectively implement caching and prefetching strategies.

3.2.3 The Application Related Component within the Wired Network

This component accesses data located on hosts within the wired network and contains functions to reduce the volume of data transferred to the mobile device. In this component application specific information can be used.

In many cases a further decomposition of this component will be useful. For instance a supervising component may create various subcomponents which perform subtasks. These subcomponents migrate independently to servers to communicate locally. The supervising component collects the results of all subcomponents, processes these results and sends only a summary to the mobile device after filtering. In general the decomposition of this component depends on the task of the application and the distribution of accessed servers and information resources.

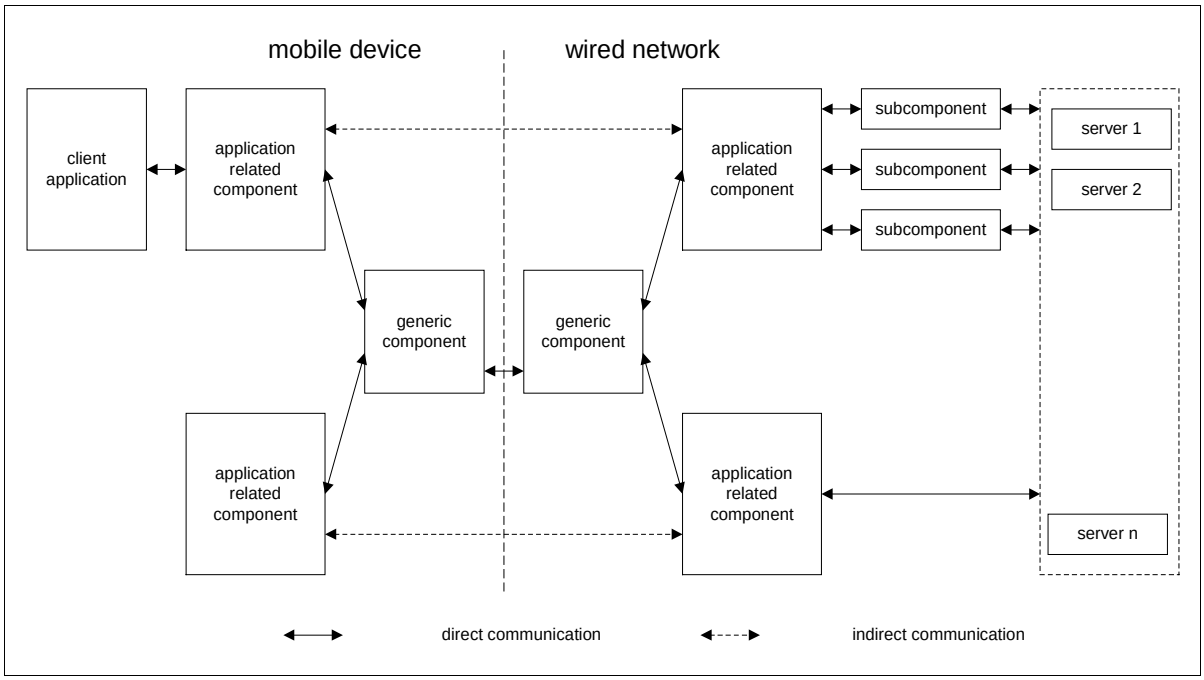


Figure 1: A generic partitioning model

3.2.4 The Generic Components (Generic Proxy)

The pair of generic components mediates the communication between the other components. While the other components exist per application, these two only exist per mobile device, Thus they are used by all applications of a mobile device.

In our concept the application related components are intended to be independent from the generic proxy. That means that all application components are also able to communicate without the generic proxy. This is useful if a high bandwidth connection to the mobile device is available.

It should be possible to dynamically bind and unbind application components to the generic proxy according to changes in the execution environment.

In a generic component incoming messages are inserted into a queue for further processing. All messages are encapsulated in an uniform data container which contains a description of the included data. The description contains routing information, data type, two id's to identify data streams and packets within a stream and information how the packet should be handled by the generic proxy. This allows each application to configure the handling of its data in the generic proxy. For instance an application can determine if a packet should be compressed and which minimal quality is tolerable for the compression. Packets are exchanged as single messages. Therefore each agent contains one method to receive messages (which is similar for all agents) with a data object as argument.

The queuing mechanism is not determined in the model. Currently we use a priority queue in our prototype application. Therefore the user can specify a priority for each application according to time constraints concerning the delivery of there messages. Each message is assigned with an priority belonging to its application before it is enqueued. Other queuing mechanisms are applicable. The priority could also be determined by the application itself according to the position of data in a data structure. Also multiple queues (one for each priority) could be used (see [9],[12]).

The messages remain in the queue until the transfer is possible. Selected messages are adapted to the available bandwidth by compressing and converting these according to the data type. A protocol adapted to the communication media can be used to optimize the transfer [5]. For instance multiple parallel send threads can be used for communication media with high delay times to better exploit the available bandwidth [9]. If a generic component receives a message from its twin component the message is mediated according to the included target information.

4 A Sample Application

An email application has been implemented to validate our model. It is partitioned using the described model. The Netscape Navigator Mail represents the client of the existing client/server application (the client component in figure 1). The application related component on the mobile device provides a user interface to configure and control all application components (agents) and to input filter data and the information required to access the email account. New messages are delivered to the mail server. The application specific component in the wired network is called email agent. It migrates to the mail server host or a host close to the server. Once there it will ask for new messages using the information for the email account. New messages are filtered in two ways using the header information (e.g. sender, receiver or subject) and information in the body (e.g. body text or attachment data type and size). The filtered messages are sent to the component at the mobile device. Therefore the generic proxy is used.

The implementation is based on the ObjectSpace Voyager core technology API [7] and the JDK 1.2 [8]. The communication between agents is based on the RMI mechanism of Java. For further details see [4].

Table 1 shows the transfer times of messages with and without the indirection over the proxy agent. We conducted our measurements for 100VGAnyLAN on two 200MHz Pentium Pro, 64MB RAM workstations running WindowsNT and for Ethernet and the wireless LAN on a ThinkPad 760D (Pentium 166MHz, 64MB RAM) running Windows95 and one of the workstations mentioned. The values are average times over 500 runs. We transferred an email containing a JPEG picture with a size of 12204 bytes in high and 4439 bytes low quality. The decoding and encoding needs an average of 135,6 ms altogether.

The values for 100VGAnyLAN and Ethernet are highly influenced by the serialization/deserialization times. The use of the proxy components only is advantageous for disconnection handling. Using the wireless LAN, the transfer time of the compressed data is significantly lower than the times for transferring uncompressed data. The indirection over the generic proxy takes also some time but the benefit for disconnection handling and adaptation justifies this effort.

		100VGAnyLAN	3Com Ethernet	Xircom Netwave wireless LAN
with generic proxy	low quality	180	179,2	444,7
	high quality	190,8	189,1	809,6
without generic proxy	low quality	55,5	57,3	378,3
	high quality	58,8	72,1	774,4

Table 1: Message transfer times in ms

5 Conclusion

The partitioning model described in this paper enables the dynamic adaptation of applications to dynamic changes of their execution environment. It introduces two component pairs. These components contain application related and generic functions to handle disconnections and variations in connection quality. Generic and application specific techniques are placed in components where they can access required information to work effectively. While other approaches focus one special problems like filtering of information [5] or data reduction [6] our model integrates most of the these techniques. Another advantage is the possibility to share the generic functionality between applications. The generic proxy provides a robust software component to adapt data transfer which can be used by all applications of a mobile device and independently of underlying network protocols. The handling of the data remains independently configurable for each application. Because of the defined and limited functionality of the generic proxy, this component is small compared to the application related components. This reduces the effort necessary after the movement of the mobile device. Only the generic proxy has to be moved. This decouples the application related parts from the movement of the mobile device so own migration strategies can be applied, which includes the migration to a server near the most probable next location of a mobile user.

An important fact is the use of mobile agents as application components which facilitates a dynamic placement of application parts and the dynamic placement of the proxy during runtime. The capability of migration is performed by the agent system. The agent system also provides other services (e.g. naming and locating, security) which can be used by all applications. Mobile agents also facilitate the execution of autonomous operations as described before. To enable the adaptation to the execution environment, the environment has to be sensed by the components. Therefore the agent system must provide additional services for instance to observe the quality of network connections. By using an agent system based on Java the proxy can be used on any platform which provides a Java virtual machine. Only the availability of the agent system and the class files are special requirements in our approach.

References

1. Marc E. Fiuczynski and David Grove: *A Programming Methodology for Disconnected Operation*; Technical Report, University of Washington, March 1994

2. James J. Kistler and M. Satyanarayanan: *Disconnected Operation in the Coda File System*; ACM Transactions on Computer Systems, 10(1), pp. 3-25, February 1992
3. Terri Watson: *Effective Wireless Communication through Application Partitioning*; In Proc. of the Fifth Workshop on Hot Topics in Operation Systems (HotOS-V), May 1995
4. Alexander Schill, Albert Held, Wito Böhmak, Thomas Springer, Thomas Ziegert*: An Agent Based Application for Personalized Vehicular Traffic Management*; Lecture Notes in Computer Science 1477 Mobile Agents, Springer, ISBN 3-540-64959-X, pp. 99-111, 1998
5. Bruce Zenel, Dan Duchamp: *A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment*; The Third Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1997 pp. 248-259
6. Armando Fox and Steven D. Gribble and Eric A. Brewer and Elan Amir: *Adapting to Network and Client Variability via On-Demand Dynamic Distillation*; Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 160-170, October 1-5, 1996
7. ObjectSpace: *Voyager Core Technology User Guide, Version 2.0*, 1998
8. Sun Microsystems: *The Java Developers Kit, Version 1.2;* http://java.sun.com/products/jdk1.2/index.html, 1998
9. Alexander Schill, Sascha Kümmel, Thomas Springer and Thomas Ziegert: *Experiences with an Adaptive Multimedia Transfer Service for Mobile Environments*; Workshop Interactive Applications of Mobile Computing, Rostock, Germany, November 24-25, 1998
10. Thomas Springer: *Masters Thesis (Diplomarbeit): Verteilung von Applikationen in Systemen mit mobilen Endgeräten;* Dresden University of Technology, Department of Computer Science, Institute of Operating Systems, Data Bases and Computer Networks, 1998
11. Terri Watson: *Application Design for Wireless Computing*; In Proc. of the 1994 Mobile Computing Systems and Applications Workshop, December 1994
12. Dirk Gollnick, Sascha Kümmel, Alexander Schill and Thomas Ziegert: *Off-Line Verteilung multimedialer Daten in mobilen Systemen*; in GI/ITG-Fachtagung Kommunikation in Verteilte Systemen, Springer Verlag, pp. 357-371, February 1997

Frameworks for Managing and Understanding Mobile Agent Complexity

Economic Markets as a Means of Open Mobile-Agent Systems

Jonathan Bredin, David Kotz, and Daniela Rus
Department of Computer Science
Dartmouth College
Hanover, NH 03755
{jonathan, dfk, rus}@cs.dartmouth.edu

May 1, 1999

Abstract

Mobile-agent systems have gained popularity in use because they ease the application design process by giving software engineers greater flexibility. Although the value of any network is dependent on both the number of users and the number of sites participating in the network, there is little motivation for systems to donate resources to arbitrary agents. We propose to remedy the problem by imposing an economic market on mobile-agent systems where agents purchase resources from host sites and sell services to users and other agents. Host sites accumulate revenues, which are distributed to users to be used to launch more agents. We argue for the use of markets to regulate mobile-agent systems and discuss open issues in implementing market-based mobile-agent systems.

1 Introduction

One of the more recent items in a network programmer's tool box is code mobility. The technique is becoming more common in applications programming, network management [BPW98], video conferencing [BPR98], software distribution and installation, unreliable networked weather forecasting [Joh98], and client-server networking alternatives [Mul98].

Mobility allows the programmer to easily distribute resource usage throughout the network over time. Resource contention can be mitigated by relocating execution to less utilized machines on the network. Additionally, since mobile agents are autonomous, they may schedule their own computation at a later time at a remote host, avoiding times of congestion.

We examine some restrictions typically present in mobile-agent systems. Specifically, few mobile-agent applications implement communication with agents from other applications or submitted by competing users. This shortfall might be surprising considering that agent technology is often advertised as a unifying design paradigm nurturing both cooperation and competition.

Many of the restrictions faced by usable mobile-agent systems stem from the distributed nature of mobility: because decisions are made throughout the network, coordination becomes more difficult. Human societies have implemented economic markets as a solution for distributed control. We believe that the same solution can be applied to create societies of mobile agents.

We propose that mobile agents buy computational resources from their hosts using a scarce verifiable currency. The agents' priority would be governed by limited endowments. The danger of denial-of-service attacks is bounded by agents' expenditure. Systems can buy outside computational resources to flexibly expand their computing base. Conversely, idle resources may be sold to users from other sites. Finally, a fair price system provides valuable information that allows agents to autonomously and efficiently balance network load.

We conclude by establishing goals necessary to implement computational markets in the context of mobile-agent systems: a low-overhead verifiable currency system, efficient incentive-compatible pricing mechanisms, a set of standards conveying the rules and bases of such pricing mechanisms to allow rational planning by agents, and both demand-based and reservation-based consumption.

2 Motivation

As mobility becomes more commonly used in network programming, we observe that, so far, applications operate in a closed environment. The range of sites to which an agent can jump is severely restricted and it is common for all agents in the system to only represent the interest of a single user or cooperating group of users. Additionally, typically agents only communicate with other agents of the same application.

The number of sites to which an agent can jump is limited. Why should a host allow any agents to visit at all? If we ask the analog question in web browsing, in a large number of situations there is a clear advantage for web servers to supply their resource (information) to arbitrary clients. This information dissemination is generally done to boost the reputation of the host site's owners, clients, or products.

Computational resources exported by mobile-agent hosts are much more difficult to control. The host site generally has no assurance that an arbitrary agent's actions will have any beneficial effects. There is little incentive for a host system to provide resources to an arbitrary agent. Not only is there the additional congestion incurred through normal mobile agent use, but there are additional risks from denial-of-service attacks and other irresponsible uses of resources.

A mobile-agent system's value is not only dependent on the number of participating host sites, but also on the the number of participating users and agents. Most non-research mobile-agent applications assume that agents are all issued by cooperating owners. Typically, only one entity issues mobile agents for a task. This entity may be represented by many users working for a university or company, but the interests of the agents are generally complementary and it is in agents' best interest to cooperate. Essentially, agents in such situations can be viewed as having a common owner.

Even if all agents share a common goal, their use distributes decision-making processes throughout the network. To perform efficiently, agents must be able to coordinate and assess

the impact of their actions. Ideally, the medium for this information exchange should be fast and incur minimal overhead.

We see few real-world examples of agents coordinating with agents involved in other applications. In stark contrast, consider the World Wide Web. A user's browser sends requests to thousands of host sites to retrieve information. Much of the information retrieval is more than simply examining an HTML file. Often browsers exchange cookies with servers, negotiate security protocols, retrieve dynamically produced web pages, download applets, and forward retrieved information to other applications to be processed. A typical web browsing-session can involve the use of several dozen application programs. We have the same goal for mobile-agent systems.

3 Markets

To overcome the limitations currently experienced by mobile agents, we propose to establish an economic market for computational resources and services. Mobile agents arriving at a host site will purchase the resources necessary to complete their task. These resources could include access to the CPU, network and disk interface, data storage, and databases. Presumably, agents are providing valuable services to their users. It is possible that other agents would also benefit from the service, so agents could sell their services to users and agents. Eventually, currency will accumulate at host sites. Revenues are then distributed to local users who in turn disperse their income to their agents completing the cycle.

The currency used in computational markets does not necessarily have to be tied to legal tender currency. If the currency is exchangeable for real dollars, however, system administrators can essentially export and import their computational resources. Access to underutilized resources may be sold to mobile agents (resource export). If local resource contention is high, then users may launch agents to carry on computation elsewhere (resource import).

Because currency is scarce, budgets are finite, and all resource consumption is tied to expenditure, agents' lifetimes are limited. The extent of a denial-of-service attack, wanton consumption done with the intention of excluding other users from the resource, is limited. Given that resource pricing is fair, hosts will be happy to entertain a denial-of-service attack to maximize revenues. An efficient pricing policy will ensure that demand and price are positively correlated and make denial-of-service attacks extremely expensive operations, deterring offenders.

Price, the same mechanism that discourages wasteful consumption, serves as a simple metric of resource contention and site congestion. Price advertisement provides a simple means of agent coordination as follows. Revenue maximizing hosts will charge what the market will bear. High prices due to congestion give agents incentive to distribute themselves evenly throughout the network or defer execution to a less congested time. Thus a pricing system effectively implements both temporal and spatial load balancing.

The idea of selling computational resources to mobile programs is not new. We discuss a few recent implementations next. POPCORN [RN98] is a system that uses markets to distribute "computelets" through the network to take advantage of idle CPUs. The approach is intended for parallel programs where interaction among threads is limited.

The Geneva Messengers project [Tsc97] applies market ideas to allocate CPU usage and memory to visiting *messengers,* lightweight mobile programs implemented in a Postscript-like language. Host sites heuristically set prices by examining the amount of resources requested by the present messengers.

In Telescript [Whi96] agents carry *permits* to access specific host-site resources. As a permit is used, hosts trust each other to diminish the permit's power. The result is that agents' lifetimes are limited. This use of permits can be viewed as a limited form of a market. Host sites distribute permits for each resource to be controlled, though a permit for one resource is not easily convertible to another. A more general form of this mechanism would be to have a universal permit, currency, that could be exchanged for other permits.

An extreme point of view is taken in MarketNet [YDFH98] where currency-resource exchange is the exclusive form of security. Different levels of security access are sold to users. Sites may discount access to certain populations by setting a lower price in a separate currency. Presumably this new currency is unusable by users outside the group.

4 The Challenge

There are many hurdles to cross on the way to implementing an effective market system. Most obviously, there must be some means of exchange such as a secure currency system. The market should be structured in such a way to reward honest behavior, to facilitate planning. Finally, this structure should be well known to all entities participating in the market.

4.1 Accounting

All markets rely on a secure means of exchange. Without this, there is no incentive for participants to cooperate. Currency should be scarce to reinforce its value. A prerequisite to this is that currency may only be spent once, i.e. a buyer may not use the same note for more than one purchase. Ensuring this is the crux of any monetary system and can carry with it significant overhead. Electronic currency in the context of mobile-agent systems has one particular caveat: an agent's money is essentially just data, data to which the host potentially has access.

There are several micro-currencies designed to minimize the cost of transaction [GMA+96, PHS98]. If the cost of currency exchange is still too large, there are other options to take. Hosts can establish a local account for each agent or its owner. Deposits into the account are periodically made with some secure payment method. Agents then withdraw from the local accounts as they compute, trusting the host site to correctly decrement the account balance. If the deposits are small or the account is known to be used over a long period of time, then there is less incentive for hosts to overdraw the local account since the payoff for cheating once is lower than conducting further honest business.

The other option is to scale the level of resource accounting. Ideally, agents would be charged for every action they take including a precise count of the instructions executed or even the number of processor cycles used; bits sent through the network interfaces; words/milliseconds of storage; etc. Such monitoring is most likely impractical, however, due

to the cost of precise measurement. At the other extreme, agents may pay a fixed fee to execute any set of instructions. Obviously, this is also inefficient in that agents will rarely use the exact level of service for which they pay.

A happy medium between the two extremes must be found. Possibly, profiling existing applications using mobile-agent technology will give some intuition on the appropriate level of control. It is quite possible that the sorts of applications that can take advantage of mobility have similar resource requirements and one can tailor an allocation policy accommodating the majority of applications.

Extending this strategy could allow hosts to offer one of several resource packages and an appropriate billing plan to agents. Each package-accounting pair would have advantages to different groups of applications depending on preferences and resource demands.

4.2 Policy Design

There are two dynamics that drive the construction of economic policy. One is the participants' goal to maximize utility, while the other is the system's engineers need to enforce an equitable allocation. These two ideals often conflict under poorly designed policies. A simple example is when the market is allowed to create monopoly. Here a monopolist will act to maximize revenue at the expense of customers. The lesson learned is that either monopolies should be regulated or that conditions allowing the existence of monopolies should not exist.

It is the responsibility of the designer of the economic system to provide an environment in which both buyers and sellers are willing to participate. Frequently, this is facilitated by constructing mechanisms where the parties choose those actions that express their true intentions, i.e, incentive compatible mechanisms. This open honesty mitigates the cost of planning and decision making.

A well used example of an incentive-compatible policy is the sealed-bid second-price auction [Vic61] where potential buyers simultaneously submit a single bid for a good. The auctioneer chooses the winner to be the participant who submitted the highest bid, but the winner pays the highest losing bid. Generally, the optimal strategy is for a buyer to submit a bid equal to the buyer's valuation of the good.

A second issue in policy design: to enforce load balancing, a resource management policy should provide a strong correlation between the contention for a resource and its price. Possibly, this might be accomplished through setting a price that increases as the quantity consumed rises. Alternatively, an auction could be held for the resource and the resource owner could let the buyers compete to set the price.

Negotiation of price is not sufficient for market-based resource allocation. The resource management policy must also take into account resource consumption scheduling. The system designer must decide whether to entertain reservations, consumption on demand, or some combination of the two. Users and agents would likely be willing to pay more for service guaranteed by reservations, but hosts might be able to sell larger quantities of resources on a demand based scheduler. This is an interesting and valuable issue to study as it effects planning on both the host and client sides and will likely deal with computationally complex issues. There is the additional dimension that users will be willing to pay amounts proportional to the quality of service.

4.3 Standardization

Finally, any effective policy will require that all participants are aware of the guidelines regulating the system. Sites will have to find some way of publishing their resource pricing and allocation policies for their potential users. For agents to be aware of rules either requires a single standard or some protocol for expressing market parameters.

Again, with this issue, there should be moderation. The point of an "agent" is to shield the user from all the intricacies of computation by providing an abstraction. The user should be aware of the service quality they receive as well as a general level of congestion of the requests to their agents, but it is the agents' responsibility to efficiently perform the task.

5 Conclusion

We believe that markets are the proper tools to provide an open mobile-agent system. They enforce an additional level of security and give incentive for agents to autonomously balance the computational load across the network. Allowing the currency used to buy computational resources to be exchanged for legal tender allows system administrators to temporarily expand their domain by importing resources as well as capitalize on idle resources by exporting them.

Implementation of a mobile-agent computational market will require further research in electronic currency exchange to minimize the overhead of currency validation. Furthermore, careful decisions must be made on the part of market designers and host-site owners to equitably distribute resources among mobile agents and local users while attempting to maximize revenue. Finally, regardless of the resource-allocation policy, for agents (or their programmers) to plan appropriately, they must be able to detect which policy is being used. Policy discovery will likely require either a single standard or a language to describe market protocols.

Establishing markets will achieve distributed decision making in mobile-agent systems. We feel that markets are natural solutions to mobile-agent coordination and resource control and will eventually allow mobile agents to be used in an open multi-application environment, though much work remains to be done to implement a working system.

Acknowledgments

This work is supported in part by the Navy and Air Force under contracts ONR N00014-95-1-1204 and MURI F49620-97-1-0382, Rome Labs under contract F30602-98-C-0006, and DARPA under contract F30602-98-2-0107.

References

[BPR98] Mario Baldi, Gian Peitro Picco, and Fulvio Risso. Designing a videoconference system for active networks. In *In Proceedings of the Second International Workshop, Mobile Agents '98*, pages 273–284, Stuttgart, Germany, September 1998.

[BPW98] Andrzej Bieszczad, Bernard Pagurek, and Tony White. Mobile agents for network management. *IEEE Communications Surveys*, September 1998.

[GMA+96] Steve Glassman, Mark Manasse, Martín Abadi, Paul Gauthier, and Patrick Sobalvarro. The Millicent protocol for inexpensive electronic commerce. *World Wide Web Journal*, 1(1), Winter 1996. Also in Fourth International World Wide Web Conference, December 1995.

[Joh98] Dag Johansen. Mobile agent applicability. In *In Proceedings of the Second International Workshop, Mobile Agents '98*, pages 80–98, Stuttgart, Germany, 1998.

[Mul98] Tomasz Muldner. Mobile computing at acadia university. Dartmouth College Computer Science Colloquia, 1998. slides at http://evilqueen.acadiau.ca/presentations/mobileagents.ppt.

[PHS98] Tomi Poutanene, Heather Hinton, and Michael Stumm. NetCents: A lightweight protocol for secure micropayments. In *USENIX Workshop on Electronic Commerce*, pages 25–36. USENIX Association, September 1998.

[RN98] Ori Regev and Noam Nisan. The POPCORN market— an online market for computational resources. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 148–157, Charleston, SC, October 1998. ACM Press.

[Tsc97] Christian F. Tschudin. Open resource allocation for mobile code. In *In Proceedings of The First Workshop on Mobile Agents*, Berlin, April 1997.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[Whi96] James E. White. Telescript technology: Mobile agents. General Magic White Paper, 1996.

[YDFH98] Y. Yemini, A. Dailianas, D. Florissi, and G. Huberman. MarketNet: Market-based protection of information systems. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 181–190, Charleston, SC, October 1998. ACM Press.

Emergent Behavior and Mobile Agents

Tony White
Bernard Pagurek
({tony,bernie}@sce.carleton.ca)
http://www.sce.carleton.ca/researchers/tony/index.html

Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6

Abstract

Naturally occurring multi-agent systems exhibit remarkable problem solving capabilities even in the absence of centralized planning. These systems exhibit complex, emergent behavior that is robust with respect to the failure of individual agents. Such systems are usually characterized by the interaction of a large number of simple agents that sense and change their environment locally. In this paper, we discuss mobile agents and how they represent a novel problem solving paradigm that can exploit naturally occurring multi agent system (biological) metaphors. We introduce the principle of synthetic ecologies of chemically inspired agents in order to model problem solving behavior in networks. We demonstrate the utility of the architectural principles by describing example applications for problem solving in the communications domain.

Keywords: mobile agents, agent coordination, agent collaboration, swarm intelligence

1. Introduction

The advantages of mobile agents have largely been discussed in terms of technology [Chess et al., 97] and the value of individual agent autonomy (see, for example, [Bieszczad et al, 98]). It is possible to view them as an approach to problem solving where mobility and interactions with the network locally are stressed. Similarly, coordination mechanisms for mobile agents have been discussed in terms of blackboard-style algorithms, with the agents tending to be rational, having a knowledge of self and a goal to be achieved (see, for example, [O'Hare and Jennings, 96]). In fact, several implementations of such systems are being investigated by the mobile agent community [Picco et al., 99]. Symbolic systems of this type are often brittle, unable to cope with the failure of a single agent and may depend upon planning by a central agency in order to achieve coordination. Such systems often have to cope with the latency problems inherent in centralized systems. We believe that these limitations undermine the value and power of mobile agent systems.

It is difficult to argue against the effectiveness of naturally occurring multi agent systems and, in particular, systems exhibiting mobility. Societies of simple agents are capable of complex problem solving while possessing limited individual abilities [Franks, 89; Hölldobler and Wilson, 94]. They often possess no central coordination of activity; problem solving is distributed. Societies of such mobile agents are found at all levels of evolutionary complexity, from bacteria to ants and beyond. It is common in such societies to observe social coherence although when behavior of the individual is observed, a large stochastic component is present. Stated another way, such societies exhibit *emergent* behavior.

Problem solving by societies of simple agents has a number of common characteristics. Inter-agent communication is local; no single agent has a global view of the world. Communication is also achieved using simple signals and these signals are time dependent; e.g. they usually decay with time. Signal levels provide the driving force for migration patterns. Individual agents sense and contribute signal energy to the environment. In this description of the problem solving process, there are two distinct and important agent characteristics. First, there is the role of the agent within the problem solving process; i.e. how the work of problem solving is distributed to a *diverse* set of agents. Second the degree to which the actions of one agent reinforce the actions of other agents in the society of problem solvers.

The appeal of swarms of biologically inspired agents for industrial problem solving has recently been appreciated [Parunak, 98]. Research into the problems and potential of multiple, interacting swarms of mobile agents is just beginning [White and Pagurek, 98].

In the remainder of this paper, we briefly describe the principles of Swarm Intelligence and Stigmergy. We then use these principles as motivation for the **Synth**etic **E**cology of **C**hemical **A**gents (SynthECA) and provide arguments as to the value of the abstraction. SynthECA is then used to indicate how several interacting swarms of agents would be capable of problem solving in networks. The paper then concludes with a review of its important messages.

2. Swarm Intelligence and Stigmergy

Swarm Intelligence [Beni and Wang, 89] is a property of systems of unintelligent agents of limited individual capabilities collectively exhibiting intelligent behavior. An agent in this definition represents an entity capable of sensing its environment and undertaking simple processing of environmental observations in order to perform an action chosen from those available to it. These actions include modification of the environment in which the agent operates. Intelligent behavior frequently arises through indirect communication between the agents, this being the principle of stigmergy [Grassè, 59]. It should be stressed, however, that the individual agents have no explicit problem solving knowledge and intelligent behavior arises (or emerges) because of the actions of societies of such agents.

Individual ants are behaviorally simple insects with limited memory and exhibiting activity that has a stochastic component. However, collectively ants manage to perform several complicated tasks with a high degree of consistency (see, [Franks, 89; Hölldobler and Wilson, 94] for examples).

Two forms of stigmergy have been observed. *Sematectonic* stigmergy involves a change in the physical characteristics of the environment. Ant nest building is an example of this form of communication in that an ant observes a structure developing and adds its ball of mud to the top of it. The second form of stigmergy is *sign-based.* Here something is deposited in the environment that makes no direct contribution to the task being undertaken but is used to influence the subsequent behavior that is task related.

Sign-based stigmergy is highly developed in ants. Ants use highly volatile chemicals called pheromones (a hormone) to provide a sophisticated signaling system. Ants foraging for food lay down quantities of pheromone marking the path that it follows with a trail of the substance. An isolated ant moves essentially at random but an ant encountering a previously laid trail will detect it and decide to follow it with a high probability and thereby reinforce it with a further quantity of pheromone. Equally importantly, pheromones evaporate. The collective behavior which emerges is a form of autocatalytic behavior where the more the ants follow the trail the more likely they are to do so. The process is characterized by a positive feedback loop, where the probability that an ant chooses any given path increases with the number of ants choosing the path at previous times.

3. SynthECA

The **Synth**etic **E**cology of **C**hemical **A**gents (SynthECA) exploits ant-inspired agents to solve problems by moving over the nodes and links in a network and interacting with "chemical messages" deposited in that network. Chemical messages have two attributes, a label and a concentration. These messages are persistent and are the principal medium of communication used between both swarms and individual swarm agents. Chemical messages are used for communication rather than raw operational measurements from the network in order to provide a clean separation of measurement from reasoning. In addition, chemical messages drive the migration patterns of agents, the messages intended to lead agents to areas of the network which may require attention. Chemical labels are digitally encoded, having an associated pattern that uses the alphabet {1, 0, #}. This encoding has been inspired by those used in Genetic Algorithms and Classifier Systems [Goldberg, 89] (for example). The hash symbol in the alphabet allows for matching of both one and zero and is, therefore, the "don't care" symbol.

Agents in the SynthECA system can be described by the tuple, $\mathcal{A}=(\mathcal{E},\mathcal{R},\mathcal{C},\mathcal{MDF},m)$. This definition is described at length in [White and Pagurek, 98] and will only be briefly described here. SynthECA agents can be thought of as an implementation of the engineering principles for multi agent systems described in [Parunak, 98]. Agents in the SynthECA system can be described using five components:

- emitters ($\mathcal{E}$),

- receptors (***R***),
- chemistry (***C***),
- a migration decision function (***MDF***),
- memory (***M***)

An agent's emitters and receptors are the means by which the local chemical message environment is changed and sensed respectively. Both emitters and receptors have rules associated with them in order that the agent may reason with information sensed from the environment and the local state stored in memory. The chemistry associated with an agent defines a set of chemical reactions. These reactions represent the way in which sensed messages can be converted to other messages that can, in turn, be sensed by other agents within the network. The migration decision function is intended to drive mobile agent migration and it is in this function that the foraging ant metaphor, as introduced in [Dorigo et al, 91], is exploited. Migration decision functions have the following forms:

$$p_{ij}^{k}(t) = F(i,j,k,t) / \aleph_k(i,j,t), \quad R < R^* \qquad (1)$$
$$= S(i,j,t) \qquad \text{otherwise}$$

$$\aleph_k(i,j,t) = \Sigma_{j \text{ in } A(i)} F(i,j,k,t) \qquad (2)$$

$$F(i,j,k,t) = \Pi_p[T_{ijkp}(t)]^{-\alpha_{kp}}[C(i,j)]^{-\beta} \qquad (3)$$

$$F(i,j,k,t) = \max_j \Pi_p [T_{ijkp}(t)]^{-\alpha_{kp}}[C(i,j)]^{-\beta}, \text{ when } j = j^{max} \qquad (4)$$
$$= 0 \qquad \text{otherwise}$$

where:

$p_{ij}^{k}(t)$ is the probability that the k^{th} agent at node i will choose to migrate to node j at time t,
α_{kp}, β are control parameters for the k^{th} agent and p^{th} chemicals,
$\aleph_k(i,j,t)$ is a normalization term,
A(i) is the set of available outgoing links for node i,
C(i,j) is the cost of the link between nodes i and j,
$T_{ijkp}(t)$ is the concentration of the p^{th} chemical on the link between nodes i and j for which the k^{th} agent has receptors at time t,
R is a random number drawn from a uniform distribution (0,1],
R* is a number in the range (0,1],
S(i,j,t) is a function that returns 1 for a single value of j, j*, and 0 for all others at some time t, where j* is sampled randomly from a uniform distribution drawn from A(i),
F(i,j,k,t) is the migration function for the k^{th} agent at time t at node i for migration to node j,
j^{max} is the link with the highest value of: $\Pi_p [T_{ijkp}(t)]^{-\alpha_{kp}}[C(i,j)]^{-\beta}$.

The intention of the migration decision function is to allow an agent to hill climb in the direction of increasing concentrations of the chemicals that a particular agent can sense, either probabilistically (equation (3) for F(i,j,k,t)) or deterministically (equation (4) for F(i,j,k,t)[1]). However, from time to time, a random migration is allowed, this being the purpose of the function S(i,j,t). This is necessary, as the network is likely to consist of regions of high concentrations of particular chemical messages connected by regions of low or even zero, concentrations of the same chemicals. The addition of this random element assists an agent in escaping from local minima.

---

1. Note that $p_{ij}^{k}(t) = 1$ for $j=j^{max,}$ and 0 otherwise.

Finally, memory is associated with each agent in order that state can be used in the decision-making processes employed by the agent.

Why is the chemical abstraction important?

First, the SynthECA chemical abstraction forces us to design agent systems assuming nothing about concurrency and, as such, draws for its inspiration on the Chemical Abstract Machine (CHAM) [Berry and Boudol, 92]. CHAM provides a framework for developing specifications that does not bias the described systems towards any particular computational model. This abstraction is particularly important for mobile agents as it not possible to make assumptions with regard to action-event sequences in networks. Second, SynthECA provides for a single messaging abstraction that allows a broadcast capability through generalized receptors and facilitates signal reinforcement through concentration of a chemical. Third, agent chemistry allows energy and entropy flow through the processes of individual chemical reactions; e.g. evaporation. Reinforcement and energy flows are characteristics considered *fundamental* to systems that are to exhibit emergent behavior. Finally, analytical techniques based upon reaction kinetics and statistical thermodynamics [Millonas, 95] provide powerful tools for analysis.

4. Modeling with SynthECA

One of the goals in proposing SynthECA was that it should be capable of supporting subsumption architectures [Brooks, 91]. It was our intent to allow the addition of new layers to operational agent systems without modification of the encoded behavior of agent classes in existing layers. SynthECA achieves this goal by providing chemical signals that are passed between layers. Signals may excite or inhibit migratory behavior within layers depending upon the sensitivity to the particular chemical. For example, the reliability chemical deposited by a fault location agent provides an inhibitory signal that affects the migration of routing agents.

We have applied SynthECA to the problems of routing, fault location and planning in networks. These are briefly described in the following three subsections. A mechanism for implementing agent fault tolerance is described in a fourth subsection.

4.1 Routing

The foraging behavior of ants and their use of pheromones for route reinforcement map easily and naturally to the problem of route finding in networks. SynthECA routing agents use distinct chemicals for particular point-to-point, point-to-multipoint and shortest Hamiltonian cycle routing tasks. Each routing agent senses a routing chemical (r-chemical), a reliability chemical (R-chemical) and a quality of service chemical (qos-chemical). Routing agents are sent out from source to destination(s), dropping a quantity of r-chemical on the return path to the source node having discovered the destination. Path emergence is considered to have occurred when the majority of the returning routing agents follow the same path (see [White, Pagurek and Oppacher, 98] for more details and results). Once a route has emerged, an allocator agent traverses the path and assigns resources to the connection. A quality of service sensing agent (qos-agent) then monitors the end-to-end quality of the allocated connection. Several examples of the exploitation of the foraging behavior of ants for routing have also been reported [Schoonderwoerd et al., 97], [Bonabeau et al., 98], [Di Caro and Dorigo, 97].

4.2 Fault Location

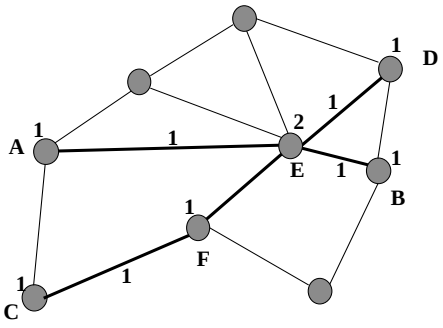


Figure 1: Fault Location Example

Whenever the connection quality sensed by the qos-agent changes significantly, another agent traverses the nodes and links involved in the connection dropping a quantity of qos-chemical proportional to the change in sensed quality of service. Given that *many* connections share network resources, more qos-chemical will be dropped on common elements than others, in many cases making possible the identification of the component responsible for the quality of service change. This is shown in Figure 1 opposite where two connections, AB and CD, are present. The numeric

labels on the nodes and edges represent the quantity of qos-chemical deposited for a hypothetical quality of service change. A qos-location-agent senses qos-chemical concentrations, constantly migrating towards higher concentrations of qos-chemical. When it reaches the peak of qos-chemical concentration, it initiates diagnostic activity on the component in order to determine the problem. In the figure on the previous page, diagnostic activity would be initiated on node E. Once the problem has been corrected, a quantity of R-chemical is dropped in proportion to the time taken to diagnose/repair the problem. The R-chemical is used by the planning and routing agents. A reliability agent (R-agent) circulates constantly within the network, hill-climbing in the space of the R-chemical. When the R-agent reaches a device for which the reliability threshold has been compromised, the device is scheduled for replacement or maintenance activity. Further details on the fault location scenario may be found in [White, Bieszczad and Pagurek, 98].

4.3 Planning

The R-chemical in the previous section may be used to drive the planning process along with a chemical resulting from network congestion (c-chemical). Arguably, the R-agent is an example of a planning agent as it reacts to the synthesis of events (quality of service changes) over an extended period. Congestion sensing agents (c-agents) circulate in the network, choosing to visit the least recently visited adjacent neighbor as a migration strategy. They interact with (non-SynthECA) agents that measure the utilization of resources on that device[2] and drop a quantity of c-chemical in proportion to the utilization of the device. Given that routing agents sense the r-chemical, and choose to avoid resources that are unreliable, they are likely to cause increasing congestion because of unbalanced network usage. A planning agent (p-agent) needs to identify regions of the network that are either high congestion or high unreliability areas. They do so by sensing R-chemical and c-chemical concentrations and initiating the re-planning of a region of the network for which a function of the two concentrations exceeds some threshold value.

4.4 Fault tolerance

Devices in networks are unreliable and agent loss must be tolerated if multi agent systems are to be made to function reliably. As this paper has demonstrated, SynthECA agents can be made to sense a number of chemicals. The fault location and planning agents of the previous two sections require that small numbers of such agents circulate constantly in the network. Hence, given unreliable transport and devices, we can expect agents to be lost.

We propose that two classes of agent $A_1=(e_1,R_1,c_1,MDF_1,m_1)$, $A_2=(e_2,R_2,c_2,MDF_2,m_2)$, share chemicals in their emitters and receptors; i.e. $e_1 \cap R_2 \neq \emptyset$ and $e_2 \cap R_1 \neq \emptyset$. One of the emitters of A_1 is used to generate a chemical that indicates when the agent was last at a given location. This same chemical is a member of the receptor set of A_2. A_2 uses the concentration of this chemical to decide whether a member of the A_1 class has visited a device "sufficiently frequently." Reactions on each device cause these chemicals to evaporate at a given rate. If A_2 reasons that the visit rate is too low, it spawns a new instance of class A_1 that then migrates autonomously. Similarly, when an instance of class A_1 visits a device, it senses its "visit frequency" chemical. If after performing whatever activity it is designed for it reasons that the visit frequency is too high, it dies. Obviously, the above algorithm can be repeated with indices reversed thereby leading to symbiotic fault tolerance for the two agent classes.

Simple extensions to multiple classes are possible.

5. Conclusions

We have proposed the exploitation of a number of ideas and principles from naturally occurring multi-agent systems in this paper in order to provide support for mobile agents as a problem solving technique. The essential characteristics of SynthECA agents are that they possess simple behaviors, reinforce and modify each others' actions through interaction with their environment by chemical messaging. Essentially, chemical messages are symbols with state. Problem solving is emergent in that routing agents are not explicitly told how to find a route and fault location agents are not instructed on network fault finding.

---

[2] Presumably, they use a measurement agent such as a Simple Network Management Protocol (SNMP) agent.

The scenarios presented are simple, but compelling, and we believe that considerable insight can be gained by experimental and analytical study of systems constructed using SynthECA agents.

Acknowledgements

We would like to acknowledge the support of the Communications Information Technology Ontario (CITO) and the National Science and Engineering Research Council (NSERC) for their financial support of this work.

Bibliography

[1] G. Beni and J. Wang, Swarm Intelligence in Cellular Robotic Systems, *Proceedings of the NATO Advanced Workshop on Robots and Biological Systems*, Il Ciocco, Tuscany, Italy, 1989.

[2] G. Berry and G. Boudol, The Chemical Abstract Machine, *Theoretical Computer Science*, 96(1), pp. 217-248, 1992.

[3] E. Bonabeau, F. Henaux, S. Guérin, D. Snyers, P. Kuntz and G. Théraulaz, Routing in Telecommunication Networks with Smart Ant-Like Agents. In *Proceedings of the Second International Workshop on Agents in Telecommunications Applications (IATA '98)*, Lectures Notes in AI vol 1437, Springer Verlag, 1998.

[4] A. Bieszczad, T. White, and B. Pagurek, Mobile Agents for Network Management. In *IEEE Communications Surveys*, September, 1998.

[5] R. A. Brooks, Intelligence Without Representation, *Artificial Intelligence*, Vol. 47, pp. 139-159, 1991.

[6] Chess. D, Harrison C., and Kershenbaum A., Mobile agents: Are they a good idea? In *Mobile Object Systems: Towards the Programmable Internet*, pages 46-48. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[7] G. Di Caro and M. Dorigo, AntNet: A Mobile Agents Approach to Adaptive Routing. Tech. Rep. IRIDIA/97-12, Université Libre de Bruxelles, Belgium, 1997.

[8] M. Dorigo, V. Maniezzo and A. Colorni, The Ant System: An Autocatalytic Optimizing Process. Technical Report No. 91-016, Politecnico di Milano, Italy, 1991.

[9] N.R. Franks, Army Ants: A Collective Intelligence, *Scientific American*, Vol. 77, 1989.

[10] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[11] P. P. Grassè, La reconstruction du nid et les coordinations inter-individuelles chez Bellicoitermes natalenis et Cubitermes sp. La theorie de la stigmergie: Essai d'interpretation des termites constructeurs. In *Insect Societies*, Vol. 6, pp. 41-83, 1959.

[12] B. Hölldobler and E. O. Wilson, *Journey to the Ants*. Bellknap Press/Harvard University Press, 1994.

[13] M. M. Millonas, Swarms, Phase Transitions and Collective Intelligence, In *Artificial Life III (ed. C. G. Langton). Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol XVII*. Reading, Massachussetts: Addison-Wesley, 1994.

[14] G. M. P. O'Hare and N. R. Jennings (eds.), *Foundations of Distributed Artificial Intelligence*, ISBN 0-471-00675-0, John Wiley & Sons, 1996.

[15] H. Van Dyke Parunak, Go to the Ant: Engineering Principles from Naturally Multi-Agent Systems, to appear in *Annals of Operations Research*. Available as Center for Electronic Commerce report CEC-03, 1998.

[16] G. P. Picco, A. L. Murphy and G-R. Roman, Lime: Linda Meets Mobility, Accepted for publication in Proceedings of the 21th International Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA.

[17] R. Schoonderwoerd, O. Holland and J. Bruten. Ant-like Agents for Load Balancing in Telecommunications Networks. In *Proceedings of Autonomous Agents '97*, Marina del Rey, CA, ACM Press pp. 209-216, 1997.

[18] T. White and B. Pagurek, Towards Multi-Swarm Problem Solving in Networks. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS '98)*, pp. 333-340, July, 1998.

[19] T. White, A. Bieszczad and B. Pagurek, Distributed Fault Location in Networks Using Mobile Agents. In *Proceedings of the Second International Workshop on Agents in Telecommunications Applications (IATA '98)*, pp. 130-141, July 4th-7th, 1998.

[20] T. White, B. Pagurek and F. Oppacher, Connection Management using Adaptive Agents. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, pp. 802-809, 12th-16th July, 1998.

```
package cps720.assignment2;


import java.util.Random;
import java.io.*;

public class CreateSurvey {

    /*
     * The question is of the form "How do your respond to this statement?
     * <the statement>
     * 1 = strongly disagree, 10 strongly agree. 5 no opinion.
     *
     * USAGE: java CreateSurvey <question> <dataFileName>
     * (the question string is not stored.)
     */
    private String theQuestion;

    private int rating;

    private final int STANDARDDEV = 3;
    private final int SAMPLESIZE = 1000;

    private Random gauss;
    private File f;


    public CreateSurvey(String aQuestion, String filename) {
        theQuestion = aQuestion;
        f = new File(filename);
        gauss = new Random();
    }

    public void questionAndAnswer() {
        BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
        System.out.println("Enter your response to the question.");
        System.out.println("The program will create a database of responses influenced by
your answer.");
        System.out.println("=====================================");
        System.out.println(theQuestion);
        System.out.println("=====================================");
        System.out.println();
        int evaluation = 5;
        try {
            evaluation = Integer.parseInt(br.readLine());
        } catch(IOException ioe) {
            System.err.println(ioe);
        } catch(NumberFormatException mne) {
            System.err.println("Using the default value, 5.");
        }
        if(evaluation > 10) evaluation = 10;
        if(evaluation < 1) evaluation = 1;
        rating = evaluation;
        System.out.println();
    }

    public void genData() {
         int temp = 0;
         try {
            DataOutputStream dos = new DataOutputStream(new FileOutputStream (f));
            dos.writeInt(rating);
```

```
            for(int i = 0; i < SAMPLESIZE; i++) {
                temp = getRandEquiv(rating);
                dos.writeInt(temp);

               System.out.print(temp);
            }
            dos.close();
        } catch(IOException ioe) {}
    }
   private int getRandEquiv(int aRating) {
     float g, bigSD, newRating;
     do {
         g = (float) gauss.nextGaussian();
         bigSD = g * STANDARDDEV;
         newRating = bigSD + aRating;
     } while (newRating < 1.0 || newRating > 10.0);
     return Math.round(newRating);
    }

    public static void main(String[] args) {
        String question = args[0];
        String fileName = args[1];
        CreateSurvey survey = new CreateSurvey(question, fileName);
        survey.questionAndAnswer();
        survey.genData();
    }
}
```

```
package cps720.assignment2;

import ptolemy.plot.*;
import cps720.assignment2.util.Convert;

/**
 * Uses PtPlot facilities to create a histogram from comma separated value
 * (csv) strings.
 * An example string is "3,9,15,20,15,8,2".
 * To use this class, ptplot.jar must be in the CLASSPATH.
 * Note also that the class cps720.assignment2.util.Convert must be
 * available.
 */
public class SurveyHistogram extends Histogram {

    private Histogram histogram;
    private PlotFrame frame;

    private String csvHistogram;
    /*
     * Creates a histogram (bar chart) from a cvs string.
     */
    public SurveyHistogram(String csvHistogram) {

        this.csvHistogram = csvHistogram;

        histogram = new Histogram();
        // The following line the bars up nicely
        histogram.setBinWidth(1.0);
        histogram.setBinOffset(0.0);
        histogram.setBars(0.5, 0.0);
        histogram.setTitle("Survey Results");
        histogram.setXLabel("Agree/Disagree Scale");
        histogram.setYLabel("Count");
    }

    /*
     * Displays the histogram.
     */
    public void displayHistogram() {

        int [] histogramValues = Convert.csvToArray(csvHistogram);


        for(int i = 0;i < histogramValues.length; i++) {
            int categoryCount = histogramValues[i];
            for(int j=0; j < categoryCount; j++) {
                histogram.addPoint(0, i);
            }
        }
       frame = new PlotFrame("Survey Results", histogram);
       frame.setBounds(100,100,500, 400);
       frame.setVisible(true);
        histogram.fillPlot();
    }

    // test
    public static void main(String [] args) {
        SurveyHistogram sh = new SurveyHistogram("3,4,5,6,5,4,3");
        sh.displayHistogram();
    }
```

```
}
```

CPS 720 Exam Readme, Fall 2001

The exam starts at 9:00 AM NOT 8:00 AM.

Last updated: Dec. 10 2001.

Note: This site is mirrored on the SCS server at http://www.scs.ryerson.ca/~dgrimsha/cps720/index.html

and at http://proton.scs.ryerson.ca.

You might want to note these links in case this ryerson site goes down at an invonvenient moment.

The Exam Response (bubble) sheets will be handed out before the exam this year. It would be a good to come 5-10 minutes before 9:00 so you can your sheet and the exam can be started on time.

Bring an HB pencil to the exam. This type works best on bubble sheets. A good eraser is nice (but dont't second guess yourself). If you are paranoid about bubble sheets you can also circle answers on the exam book. (But you must also fill out the bubble sheet.)

When writing your code, please use either an HB pencil or a pen. What I can't see, I can't mark :-).

The CPS720 exam consists of two parts. Part 1 consists of 15 multiple choice questions, worth 2 marks each. Part 2 consists of 2 programming questions, one for Aglets, one for JADE. They are worth 10 marks each.

The programming questions will be marked like essays, that is, without a detailed marking scheme. Some syntax errors are to be expected and will not count against you provided that they do not become too numerous.

The programs are both basic for each system. For the Aglet program there is some code in the multiple choice section which should remind you of the essentials. For the JADE program you might look at the following JADE constructs:

BasicBehaviour, CyclicBehaviour, action(), setup(), done(), receive(), blockingReceive(), block(), setPerformative(), getPefromative(), setContent(), getContent(), setName(), setType(), setSender(), addReceiver(), addtServices(), ACLMessage.INFORM, ACLMessage.QUERY_REF, ACLMessage.NOT_UNDERSTOOD, send(), doDelete(), addBehaviour(), getAID(), DFService.register().

See the assignments and the examples. (Of course, assignments 2 and 3 are considerably more complex.)

The multiple choice questions mostly cover general points discussed in the course. They are based on the course notes. As you know the course notes have plenty of links to more detailed and in depth materials such as PhD theses. These links are for reference. Of course, the more you follow the more you know. The lecture notes themselves represent the basic minimum.

This year's exam is somewhat similar to last year's exam. However last year's course did not include JADE. On the other hand it covered XML in more detail and also include Game Theory which is not on this year's course.

Fall 2000 CPS720 exam (pfd)

A solution to Assignment 2 (jar)

Ryerson Polytechnic University
School of Computer Science
Final Examinations, Fall 2000
CPS 720 Artificial Intelligence Topics

Examiner: D. Grimshaw Time: 2 hours Closed Book

YOUR NAME: _____

YOUR ID: _____

PART 1. MULTIPLE CHOICE. 24 Marks.
(12 questions, 2 marks each. Answer all questions in this part on your test response (bubble) sheet.

1. Current distributed systems, in contrast to mobile agent based systems, attempt to achieve “referential transparency”, Briefly, what does this mean?

A. A user on a client machine on a network does not know the location of a service she is accessing.
B. References in Java are not visible to the programmer.
C. Programs written in C in such a distributed system are not allowed to use pointers.
D. Only Java may be used to write programs for systems exhibiting referential transparency.

2. Ferber divides multi-agent systems into two broad categories, situated agents, and communicative agents. Which of the following statements best describes the difference between these two categories.

A. Situated agents are immobile; communicative agents are mobile.
B. Communicative agents are immobile; situated agents are mobile;
C. Situated agents are capable of planning ahead but communicative agents are not.
D. Situated agents live in a physical environment and have sensors. Communicative agents are virtual and do not have physical sensors.

3. According to Oshima and Lange, which of the following properties is NOT essential to an agent: reactive, autonomous, ability to learn, goal driven, temporally continuous.

A. autonomous
B. temporally continuous
C. ability to learn
D. reactive
E. goal driven

4. One reason that agents that plan (look) ahead might be more useful that agents that simply react to stimulants and remember certain states is:

A. They can test future states or situations without committing to them, avoiding costly physical backtracking.
B. An agent which plans uses less memory than one that does not.
C. Agents which plan ahead can choose the right thing to do more quickly than those that do not.
D. Planning agents can maximize the utility of their human masters using the methods of Game Theory.

5. Which of the following best describe a mobile agent on the Net?

A. It moves the computation (algorithm) to the data.
B. It moves the data to the computation.
C. It really behaves just like an Applet.
D. It is another way of sending data across the Net.

6. Todd Papaioannou claims there is an “architectural mismatch” between standard distributed systems emphasizing “referential transparency” and the computers these systems normally run on. What does he see as the basis of this mismatch?

A. Present distributed systems (e.g. RMI, CORBA, DCOM), are just Remote Procedure Calls (RPC). RPC’s are limited to UNIX systems and therefore are not suitable for the Internet which has many different kinds of servers.
B. Present distributed systems attempt to extend the Inter Process Communication (IPC) paradigm which is well matched to single von Neumann machines where processes share memory. But in distributed systems, memory is not shared.
C. Referential transparency is impossible to achieve because only copies of objects can be sent across networks. Therefore, in this respect, local programs and distributed programs can never appear to be the same to the user.
D. The basis of the mismatch is the failure to use mobile agents. With mobile agents, referential transparency can be achieved.

7. In addition to the Aglet itself, the Aglet system provides an Aglet proxy. What is the main reason for adding a proxy class to the system, rather then just having only the Aglet class itself?

A. There is no real reason other than as a convenience for the programmer.
B. A proxy class is necessary for sending and receiving messages in any agent system
C. The proxy wraps the Aglet in a security ‘blanket” which protects the Aglet’s public methods from direct access.
D. The proxy enables Aglets to become mobile.

8. Consider the following Aglet.

```
package exam;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.io.*;

public class ExamQ1 extends Aglet {
        private PrintWriter pw = null;
        private MyData md;
        private String file = "/home/dgrimsha/greeting.txt";
        public void onCreation(Object init) {
                md = new MyData("Hello, world");
                addMobilityListener( new MobilityAdapter()  {
                        public void onArrival(MobilityEvent me) {
```

```
                                        try {
                                                pw = new PrintWriter(new FileWriter(file));
                                                pw.println(md.getMsg());
                                                pw.close();
                                        } catch(IOException e) {}
                                }
                        });
                }
}
class MyData {
        private String msg;
        public MyData(String m) {
                setMsg(m);
        }
        public void setMsg(String msg) {
                this.msg = msg;
        }
        public String getMsg() {
                return msg;
        }
}
```

This Aglet code is compiled and the resulting class file loaded into a Tahiti Server. Using the server's Dispatch command, an attempt is made to move it to a second Tahiti server. Which of the following statements best describes what happens?

A. The Aglet goes to the second Tahiti server and writes the "Hello world" string to the file system of the machine hosting the second Tahiti server.
B. The Aglet goes to the second Tahiti server where it throws a security exception because it does not have the appropriate write permissions.
C. The Aglet refuses to move from its home server because the PrintWriter object is not serializable.
D. The Aglet refuses to move from its home server because the data it is carrying is not serializable.

9. In the Sequential Itinerary pattern the Aglet's mobility (its movement among servers) is handled by a separate class, rather than being handled by the Aglet class itself. What advantage does this separation confer?

A. Flexibility. For example, different itineraries could be "plugged into" the same Aglet.
B. Portability. The Aglet can run on more types of servers.
C. Speed. Because the Aglet does not have to handle its travel plans, it can move more quickly from one server to another.
D. Security. The separation of the itinerary from the task for an Aglet increases security of the Aglet system.

10. Supplying a DTD with an XML document allows the parser to,

A. Be smaller in size and faster in operation.
B. Catch more syntax errors.

C. Check the validity of an XML document.
D. Create a DOM from the document.

11. There are two standard parsers for XML, DOM and SAX. Which of the following statements best describes the main difference between these parsers?

A. A DOM parser converts the XML text document into a tree structure stored in memory. The SAX parser generates events from the XML document and by default stores nothing at all.
B. A DOM parser generates events and stores in memory whatever parts of the XML document the programmer chooses. The SAX parser creates a data structure in memory and then generates events from each node in that structure.
C. There is no significant difference from the programmer's point of view. The SAX parser uses events to generate a tree data structure corresponding to the original XML document. The DOM parser generates the same kind of tree structure but is not event driven.
D. The DOM parser always uses a DTD whereas with the SAX parser you do not have to use a DTD except under special circumstances.

12. The Prisoner's Dilemma is a very famous example from Game Theory. Which of the following statements best describes its significance.

A. It shows that cooperation can never be achieved because people are too selfish.
B. It shows that cooperation can be achieved if the parties are prepared to take the chance of being "suckered".
C. It demonstrates that restaurants in tourist areas are of lower quality than neighbourhood restaurants because tourists never come back.
D. It explains why Communism lost the Cold War.

PART 2. SHORT ANSWER. 12 Marks.
Answer any two (2) of the questions in this part. (6 marks each)

1. Discuss why use of mobile agents might, in some situations, reduce network traffic compared to other methods used to construct distributed systems.
2. There are at least three ways of arranging data and knowledge in distributed systems, client-server, code on demand (e.g. Applets), and mobile agents. Compare these three systems, briefly.
3. Speech Act theory divides the generation of speech into three steps (intention, generation, synthesis), and divides the decoding of speech into four steps (perception, analysis, disambiguation, incorporation). Briefly explain (or define) any six of these steps.
4. Interpreting this table as the payoff matrix for a zero sum game with three strategies for each player, answer the following questions. (The Row player wants the highest possible score, the Column player wants the lowest possible score.)

8	4	5
3	7	6
5	6	10

A. If the players played pure strategies (using minimax and maximin) what would they be for each player?
B. Should they stick with the pure strategies found in part A, or should they follow a mixed strategy? Explain your answer.

5. Consider this DTD, toolbox.dtd

```
<!ELEMENT toolbox (screwdrivers*, pliers*,saws*) >
<!ELEMENT screwdrivers (#PCDATA) >
<!ATTLIST screwdrivers type  (regular|philips|robinson) "regular">
<!ATTLIST screwdriver price CDATA #REQUIRED>
<!ELEMENT pliers (#PCDATA)>
<!ATTLIST pliers  price  CDATA #REQUIRED>
<!ELEMENT saws (#PCDATA)>
```

Write an XML document which is valid according to this DTD. Include at least one screwdriver of type, philips. Use any names (or descriptions) you like for the objects in the toolbox. You can start your XML file with,

```
<?xml version="1.0"?>
<!DOCTYPE toolbox SYSTEM "toolbox.dtd">
```

PART 3. AGLET PROGRAM. 10 Marks.
(1 Question for 10 marks.)

1. Write an Aglet which is to be dispatched from your Tahiti server to a remote Tahiti server. On the remote server a stationary Aglet is running. This Aglet has put a property named “prof” in its context. The value of this property is the proxy for this Aglet. Your Aglet, on arriving at the remote site sends a message “hello” to the stationary Aglet. The “hello” message contains a String object saying, “All is well”. The stationary Aglet replies with a message named “response” containing a String object saying “Message received”. When your Aglet receives this message, it should dispose of itself. Otherwise it should do nothing.

   You do not need to create a “master” Aglet for your server. Just assume you dispatched your Aglet using the dispatch button on your Tahiti.

Agents: Natural and Artificial

Agents and Agency

The word 'agent' is widely used. Everyone knows what it means. We have travel agents, real estate agents, FBI agents, secret agents, double agents. Tom Cruise has an agent. So does Margaret Atwood. Sometimes the word 'broker' refers to a kind of agent, for example, a stock broker.

But to actually define what the word, agent, means is not so easy. (Similar problems occur with other concepts such as intelligence, or life.) There seem to be almost as many definitions of agent, or agency as there are people trying to define these concepts!

So we will look at a few of these attempts at definition. Take your pick or make up your own.

Well, there's always the dictionary.

What does a dictionary say?

Human Agents

Another way to get at what the word, agent, means, we can try to see what characteristics human agents have in common.

Try making a list of characteristics.

Other animals

Is a dog an agent? An ant? If these are agents too, then what does that acknowledgement add to our understanding of agency

Some suggestions

Machines

Computer scientists have also taken to the word, agent. The AI community took it up first, but agents are now appearing in the context of networks and simulations.

Of course, the most obvious AI agent artifact is the robot. Robotics is one of the oldest branches of AI. Take CPS607!

Classic AI

Recent textbooks in artificial intelligence [Nilsson], [Russell & Norvig], use the idea of agency as a unifying theme. AI agents are usually individual 'creatures' living in an environment with which they interact in some way. Such agents are autonomous to a more or less extent. Most AI agents do not interact with other agents in the same environment. And, they are supposed to have some kind of intelligence.

Although many AI agents exist only in software, the AI agent paradigm is best illustrated by the robot interacting with a real world environment.

Distributed AI (DAI)

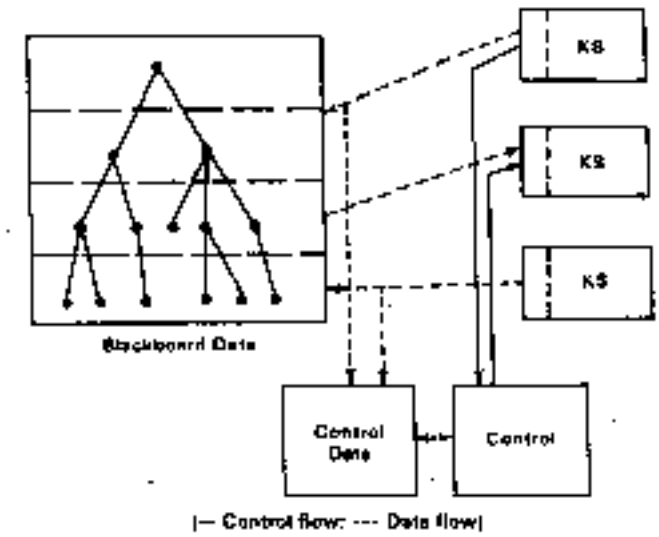
DAI has existed for many years. Practitioners of DAI consider that Intelligence cannot exist in isolation. These people hold that intelligence is, in some way, fundamentally social.

The systems of DAI usually consist of a small group of specialist agents which cooperate to solve problems. This model imitates the way teams of human specialists work together.

The classic example of DAI architecture is the so-called blackboard system, first developed in the 1970's in a system called Hearsay which was used for speech recognition.

In the blackboard model the agents are called knowledge sources. The "blackboard " is a global memory accessible to all the agents. It contains the current state of the problem. Actions by the agents gradually modify the data structures on the blackboard so that (hopefully) they come to represent the solution state of the problem.

In 19th century Germany a child was left to die in the forest But he grew up by himself (helped by wolves?) and later was found and brought back to civilization. He was given the name Kaspar Hauser. A large number of books have been written about this case. And also there are at least two films .



The "blackboard" is a metaphor. Agents communicate with one another by "writing on the blackboard" as humans might do in a brainstorming session.

Agents and the Internet

The other community some of whose members are interested in agents is the Internet community. In this case the agents exist in software environments maintained by servers on various Internet hosts. Mobility is of considerable interest. These software agents ('softbots') can move from host to host.

Also of considerable importance is inter-agent communication. Most Internet agents have more or less elaborate message passing mechanisms. Intelligence for Internet agents is not a particularly high priority, Network agents must be small if they are going to be mobile. The cannot afford to travel with the baggage of a large AI program.

Agents and Artificial Life (alife)

Another kind of multi-agent system derives from the worlds of Artificial Life. These agent systems contain large numbers of very simple identical (or almost identical) agents interacting locally in a common environment. The environment is often a finite state automaton. Such agent systems are often used to simulate aspects of human societies. For example, see the Ascape API.

Multi Agent Systems (MAS)

Distributed AI is now often referred to as Multi-Agent Systems. Most software systems are software based but there area also robotic multi agent systems involving small numbers of robots. See RoboCup.

So What is an Agent?

Next we consider three different points of view on what constitutes an agent. Click next to continue.

[top] [previous] [next] Questions?

Definition of Agent

Ferber's General Definition of Agency

An agent is a physical or virtual entity

1. which is capable of acting in an environment.
2. which can communicate directly with other agents.
3. which is driven by a set of tendencies (in the form of individual objectives or of a satisfaction/survival function which it tries to optimize).
4. which possesses resources of its own.
5. which is capable of perceiving its environment (but to a limited extent).
6. which has only a partial representation of its environment (and perhaps none at all).
7. which possesses skills and can offer services.
8. which may be able to reproduce itself.
9. whose behaviour tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representation and the communications it receives.

Note that agents are capable of acting, not just reasoning. Actions affect the environment which, in turn, affects future decisions of agents.

Autonomy

A key property of agents is autonomy. They are, at least to some extent, independent. They are not entirely pre-programmed but can make decisions based on information from their environment or other agents.

One cans say that agents have "tendencies". Tendencies is a deliberately vague term. Tendencies could be

individual goals to be achieved, or the optimization of some function.

Mobility

It is interesting that Ferber does not include mobility as a possible property of agents. Lange and Oshima do discuss this property, which although optional, certainly characterizes some agent systems.

It is interesting to note that in the natural world, agent intelligence is always associated with agent mobility (animals). Other living things (plants) have no intelligence.

Multi-Agent Systems

The term multi-agent system (MAS) is applied to systems comprising the following elements.

1. An environment E, that is, a space which generally has volume.
2. A set of objects, O. These objects are situated, that is to say, it is possible at a given moment to associate any object with a position in E.
3. An assembly of agents, A, which are specific objects (a subset of O), represent the active entities in the system.
4. An assembly of relations, R, which link objects (and therefore, agents) to one another.
5. An assembly of operations, Op, making it possible for the agents of A to perceive, produce, transform, and manipulate objects in O.
6. Operators with the task of representing the application of these operations and the reaction of the world to this attempt at modification, which we shall call the laws of the universe.

There are two important special cases of this general definition.

Purely situated agents

An example would be a robot. In this case E, the environment, is Euclidean 3-space. A are the robots, and O, not only other robots but physical objects such as obstacles. These are situated agents.

Pure Communication Agents

If A = O and E is empty, then the agents are all interlinked in a communication networks and communicate by sending messages. We have a purely communicating MAS.

We have the following definitions for these special cases.

Purely Communicating Agent

By comparison with the general definition of an agent, a purely communicating agent (or software agent) is defined as a computing entity which

1. is in an open computing system (assembly of applications, networks, and heterogeneous systems),
2. can communicate with other agents,
3. is driven by a set of its own objectives,
4. possesses resources of its own,
5. has only a partial representation of other agents,
6. possesses skills (services) which it can offer to other agents,
7. has behaviour tending towards attaining its objectives, taking into account the resources and skills available to it and depending on its representations and the communications it receives.

These types of agents are the primary focus of cps720.

Purely Situated Agent

A purely situated agent is defined as a physical entity (or perhaps a computing entity if it is simulated) which

1. is situated in an environment,
2. is driven by a survival/satisfaction function,
3. possesses resources of its own in terms of power and tools,
4. is capable of perceiving its environment (but to a limited extent),
5. has practically no representation of its environment,
6. possesses skills,
7. can perhaps reproduce,
8. has behaviour tending to fulfill its survivor/satisfaction function, taking into account the resources, perceptions and skills available to it.

A purely communicating agent is distinguished from the concept of agent in general because,

- It has no perception of other agents
- Its tendencies take on the appearance of objectives.
- It does not act in a normal, spatial environment, but rather in a computer network.

A purely situated agent do not usually communicate directly, via messages. They react to one another via preceptors (sensors) and through changes to the environment.

Aglets, which are discussed later in the course are communicating agents (for the most part). JADE agents are also primarily communicative. Ascape agents, which live on a cellular automaton, are more like situated agents, although they exist only in a software world.

[top] [previous] [next] Questions?

Agent Perspectives

Today there appear to be two groups using the concept of agent as important tool for designing and implementing software. The first is the Artificial Intelligence community. The second is the Internet community.

Agents and AI

Classic AI

Recent textbooks in artificial intelligence [Nilsson], [Russell & Norvig], use the idea of agency as a unifying theme. AI agents are usually individual 'creatures' living in an environment with which they interact in some way. Such agents are autonomous to a more or less extent. Most AI agents do not interact with other agents in the same environment. And, they are supposed to have some kind of intelligence.

Although many AI agents exist only in software, the AI agent paradigm is best illustrated by the robot interacting with a real world environment.

Distributed AI (DAI)

DAI has existed for many years. Practitioners of DAI consider that Intelligence cannot exist in isolation. These people hold that intelligence is, in some way, fundamentally social.

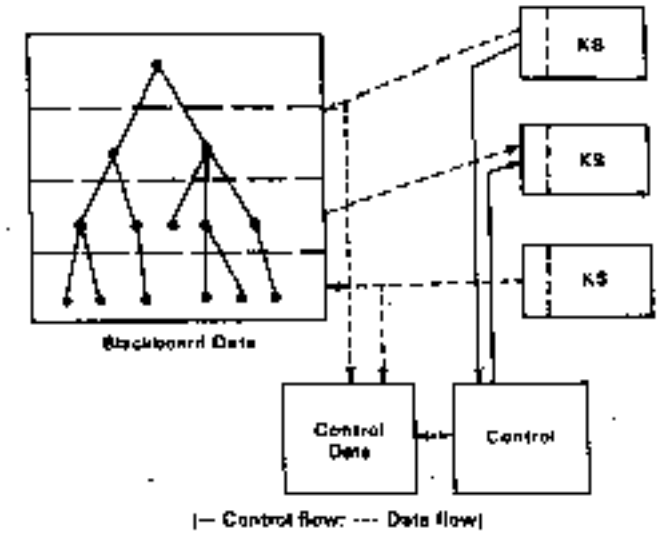
The systems of DAI usually consist of a small group of specialist agents which cooperate to solve problems. This model imitates the way teams of human specialists work together.

The classic example of DAI architecture is the so-called blackboard system, first developed in the 1970's in a system called Hearsay which was used for speech recognition.

In the blackboard model the agents are called knowledge sources. The "blackboard " is a global memory accessible to all the agents. It contains the current state of the problem. Actions by the agents gradually modify the data structures on the blackboard so that (hopefully) they come to represent the solution state of the problem.

In 19th century Germany a child was left to die in the forrest. But he grew up by himself (helped by wolves?) and later was found and brought back to civilization. He was given the name Kaspar Hauser. A large number of books have been written about this case. And also there are at least two films .(1 and 2 and 3).

The "blackboard" is a metaphor. Agents communicate with one another by "writing on the blackboard" as humans might do in



a brainstorming session.

Agents and the Internet

The other community some of whose members are interested in agents is the Internet community. In this case the agents exist in software environments maintained by servers on various Internet hosts. Mobility is of considerable interest. These software agents ('softbots') can move from host to host.

Also of considerable importance is inter-agent communication. Most Internet agents have more or less elaborate message passing mechanisms. Intelligence for Internet agents is not a particularly high priority, Network agents must be small if they are going to be mobile. The cannot afford to travel with the baggage of a large AI program.

Agents and Artificial Life (alife)

Another kind of multi-agent system derives from the worlds of Artificial Life. These agent systems contain large numbers of very simple identical (or almost identical) agents interacting locally in a common environment. The environment is often a finite state automaton. Such agent systems are often used to simulate aspects of human societies.

[top] [previous] [next] Questions?

Lange and Oshima on agency

End user perspective

An agent is a program whichs assists people and acts on their behalf. Agents function by allowing people to delegate work to them.

System perspective

An agent is a software object that

- is situated within an execution environment
- possesses the following mandatory properties
  - Reactive: senses changes in its environment and acts according to those changes
  - Autonomous:has control over its own actions
  - Goal driven: is proactive
  - Temporally continuous is contnuously executing
- and may possess any of the following orthogonal properties
  - Communicative: able to communicate with other agents
  - Mobile: can travel from one host to another
  - Learning: adapts according to previous experience
  - Believable: appears believable to the end user.

[top] [previous] [next] Questions?

Igor Nikolic's view

(quoted from his web site)

What is an Agent:

An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives

For clear understanding of the definitions, several points must be further elaborated.

Agents are:

- clearly identifiable problem-solving entities with well defined boundaries and interfaces.
- situated (embedded) in a particular environment; they receive inputs related to the state of their environment and they act on the environment through effectors.
- designed to fulfill a specific purpose; they have particular objectives (goals) to achieve.
- autonomous; they have control both over their internal state and over their behaviour.
- capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives; they need to be both reactive (able to respond in timely fashion to changes that occur in their environment) and active (able to act in anticipation of future goals).

Autonomy vs. Agency (Agents and Objects)

The point about agent autonomy need further clarification. Having control over their own behaviour is one characteristic that distinguishes Agents from Objects. While Objects have both states and behaviours, they do not contain behaviour activation or action choice. In object oriented terminology, an object may invoke any publicly accessible method on any other object at any time. Once the method is invoked, the corresponding actions are performed. In this sense, objects are totally obedient to one another, and do not have autonomy over their choice of action.

However, Parker argues that the relation between agency and autonomy is not necessary a direct one. It is arguable that the term Agent denotes an aggregation of objects that are defined more by their useful boundaries than by the definition presented above. Therefore in the following discussion the term Agent will contain both the autonomous and the odject-aggregate type. For the practical matter of modelling this distinction is not a crucial one.

According to Parker the most compelling argument for Agent Based Modeling (ABM) can be summed up as follows: ***Why don't we model it as it is in the real world?***

It basically means that ABM is useful because it offers us a possibility to create models that do away with generalisations and allow us to explore the world through discrete objects and their interactions. This by itself allows for a far richer modelling context.

Comment by D.G.

Nikolic's view of agents seems derived from the idea of agents as representatives (or servants) of people, for example, travel agents. Although autonomous with regard to how to carry out a task, the task itself is generated outside the agent (by another agent).

On the other hand, Ferber's definitions are more general, with more emphasis on autonomy. In some ways, Ferber derives his concept of agent from the animal world. His agents are more independent minded.

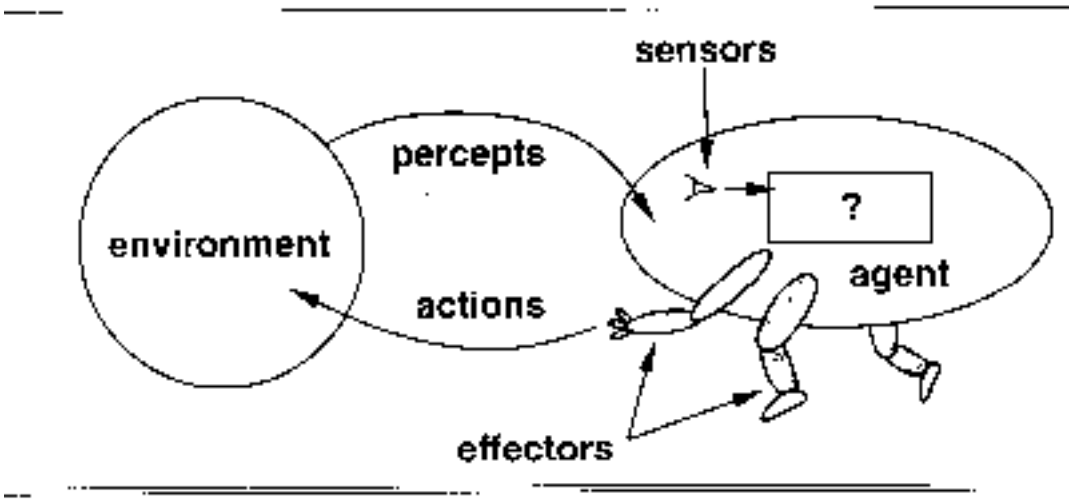
Nikolic's perspective is that of a modeller of natural systems. His thesis uses the Ascape agent system to model the spread of genetically modifed seeds.

[previous] [next]

Rational Agents

[From AI a Modern Approach by Russell & Norvig.]

A siituated agent and its environment is shown schematically in this diagram:



ıre 2.1 Agents interact with environments through sensors and effectors.

This is a robot, a situated agent. Of course there also softbots in virtual worlds. Situated agents are the type most often considered in AI. But communicating (internet) agents have a lot in common with situated agents as we saw before.

Rational Agency

The agent 'examines' its environment and the takes 'appropriate' action.

- **A rational agent does the right thing**. It carries out its task successfully. Or, at least, takes actions which, given its knowledge of its environment, **maximizes its chances** of success.

To judge an agent's effectiveness you need some kind of performance measure.

What is needed to judge the rationality of an agent

Rationality is bounded. It would be unfair to judge an agent on criteria it has no hope of achieving. For example, you can't blame a dog for not behaving like a person.

To judge the rationality of an agent fairly we need to know a number of things.

- The performance measure that defines degrees of success
- Everything that the agent has perceived so far, its **percept history.**
- What the agent knows about its current environment.
- Actions available to the agent.

An ideal rational agent

For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of evidence provided by the percept sequence and whatever built-in knowledge the agent has.

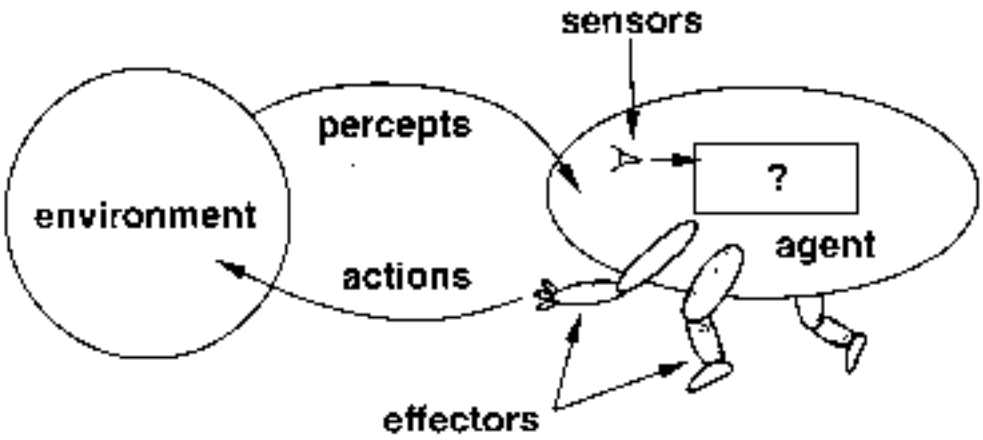
Mapping from percept (sequences) to actions

An agent's program (or brain) maps percept sequences to actions. Ideally, there should be a mapping from every possible precept sequence to every possible appropriate action. In most cases such a complete mapping is impossible. Only a superhuman programmer could anticipate every situation in advance.

In simple cases you might consider a lookup table containing an action for each perception. Such tables are usually unmanageably large. Various other methods have been used for these mappings in order to avoid this computational explosion.

- analytical functions
- production rules
- trained neural nets
- fuzzy sets

In this course we will look briefly at production rules. (More on these in cps820.)



ure 2.1 Agents interact with environments through sensors and effectors.

Agent autonomy

Artificial Intelligence is concerned with so-called autonomous agents.

- **Deterministic agent**. The robots used in automobile manufacture on assembly lines would be an example of a totally non-autonomous, deterministic robot. These are of little interest in AI.
- **An autonomous agent** (?). Chess playing programs could be considered to be autonomous agents. The good ones can easily outplay the programmers who made them. The best are as good as the best human players.

Autonomy and Intelligence

Clearly the deterministic robots are completely dumb. Every action has been pre programmed by their programmers and designers. On the other hand, in their limited field of action, chess playing programs can appear extremely intelligent. Clearly, autonomy and intelligence are closely related.

Autonomy, a fuzzy concept.

Philosophers and theologians have, from time to time, questioned whether any agent is truly autonomous. Do even humans have truly free will? Is the future predetermined? In other words, is autonomy an illusion of the human consciousness.

These are all deep questions, too deep for cps720!

A Dictionary definition

[Merriam-Webster]

Main Entry: au·ton·o·my

Function: noun
Inflected Form(s): plural -mies
Date: circa 1623
1 : the quality or state of being self-governing; especially : the right of self-government
2 : self-directing freedom and especially moral independence
3 : a self-governing state

An Agent Classification

Russell & Norvig classify agents into 4 categories, from least to most complex.

Simple Reflex (Stimulus-response) Agents

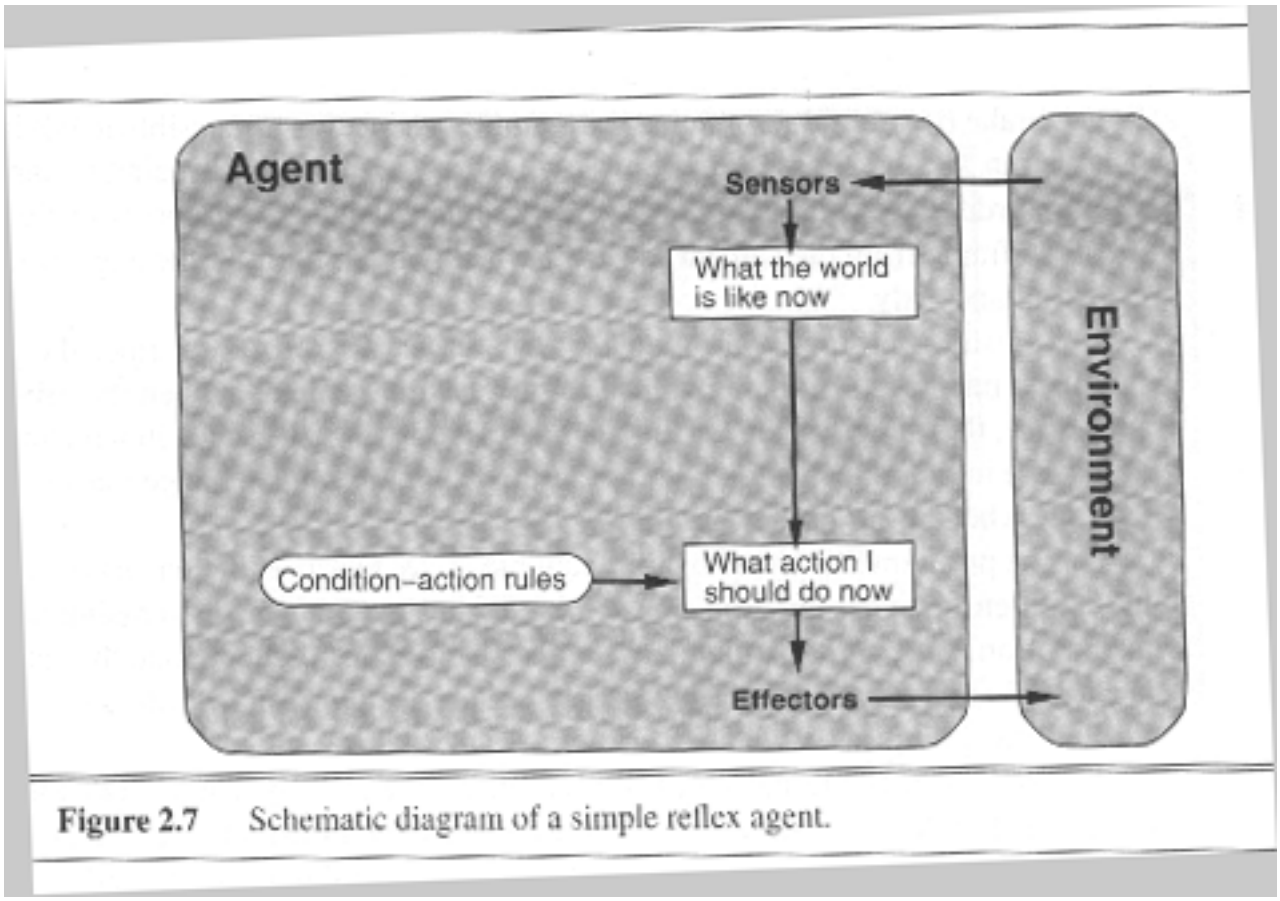


Figure 2.7 Schematic diagram of a simple reflex agent.

These agents react to immediate stimuli. They have no memory at all. The percept sequence is just the immediate environment sensed by the agent's sensors.

Nevertheless, they are capable of behaviour of considerable complexity, especially if they are part of a Multi-Agent system.

These agents are also referred to as stimulus-response agents [Nilsson]. We look at this type of agent in more detail later.

Note the "condition-action rules" in the diagram. These are production rules.

Other representations are also possible.

Reflex agent with state

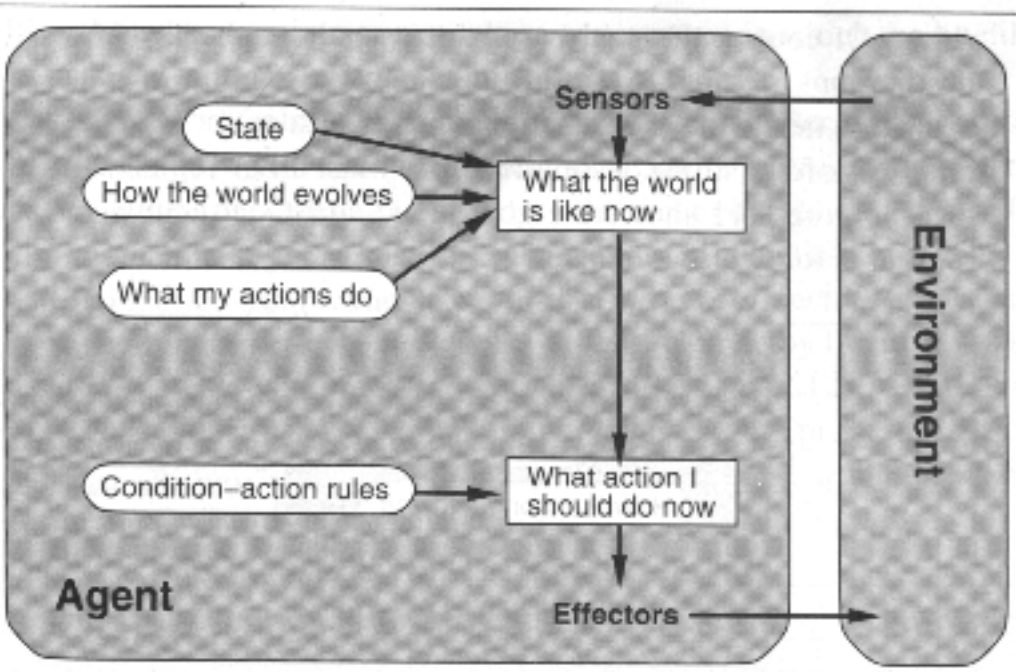


Figure 2.9 A reflex agent with internal state.

These agents have a memory of state. They can remember earlier experiences. These can be combined with current information from sensors to produce a more sophisticated response to the environment.

Goal based agents

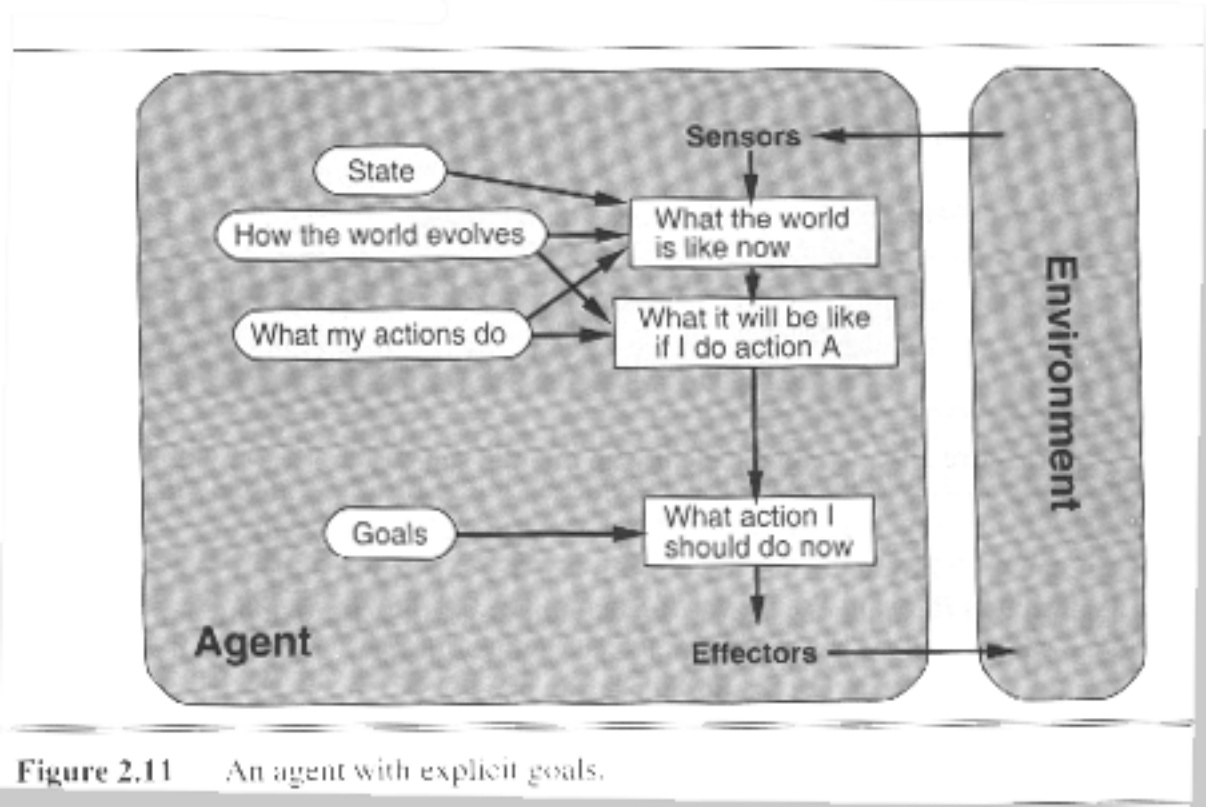


Figure 2.11 An agent with explicit goals.

Goal based agents can **plan** ahead before making their actual 'move' in their environment. They use various more or less sophisticated search methods to **search** a state space of potential future environments, as a chess player might do in his head before actually moving a piece. Future potential 'positions' are evaluated as to how close they are to the goals of the agent.

Utility Based Agents

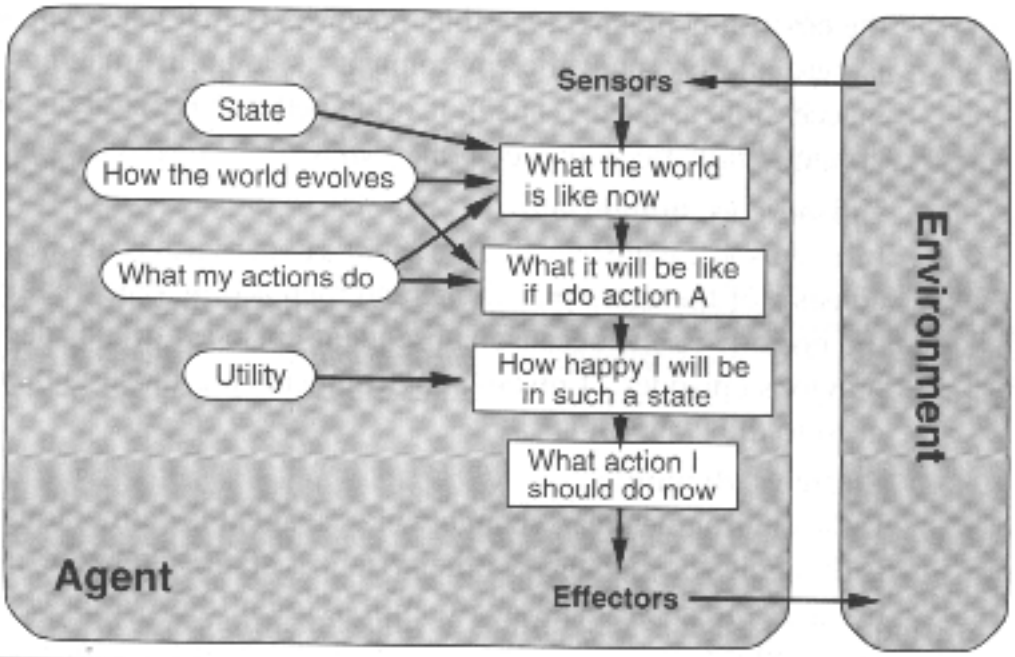


Figure 2.12 A complete utility-based agent.

At this level agents have acquired human aspects. Utility is an economic term (from micro economics) related to the concepts of happiness and personal preferences. This level of ability is beyond the present capabilities of present AI.

[top] [previous] [next] Questions?

Stimulus-Response Agents

[This is Nilsson's term. They are equivalent to Russell & Norvig's basic reactive agents.]

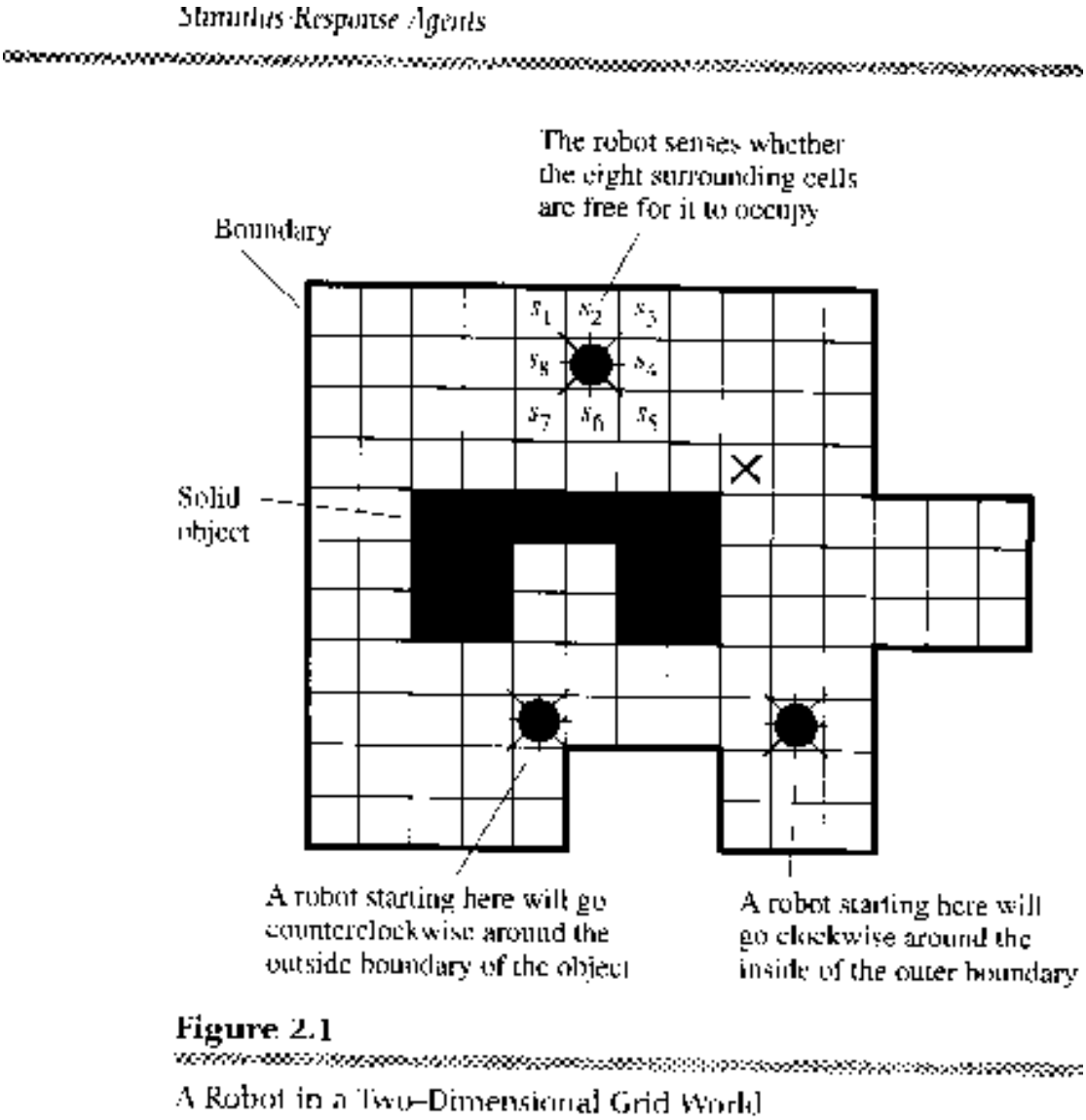
These agents, although very simple, are capable of surprisingly complex behaviour. One is reminded of the behaviour of certain simple living forms such as insects.

We discuss these agents in some detail in order to make our discussion of agents more concrete, less abstract.

These notes use Nilsson's example from chapter 2 of Artificial Intelligence, a New Synthesis.

A wall following robot

Nilsson uses the example of a simple wall following robot.



To keep things simple there are no "tight spaces". For example, the alcove in the wall in the middle of the diagram is two squares wide, not one.

Robot sensors

The robot can sense if any of its neighbouring 8 cells are free. The sensor inputs are represented by 8 binary values, s1 .. s8. These have values 0 if the corresponding cell can be occupied by the robot on its next move (free cell), and 1, otherwise. For example, if the robot is on the square marked X, then its sensors input (0, 0, 0, 0, 0, 0, 1, 0).

Robot effectors

The robot can move North(up), East(right), South(down), or West(left), (but not diagonally) into an empty square. If the target square is occupied (part of a wall), nothing happens.

The designer's job

The given

In the example, the task of the robot is to follow a boundary (wall). Also given are the robots sensor abilities and its effector capabilities.

The goal

Specify a function which maps from the robots sensor inputs to effector actions appropriate for its tasks or goals.

The design

It is common to divide the design into two parts.

- Perceptual processing phase. The sensory inputs are mapped to a feature vector
- Action processing. The effector actions are a function of the feature vector.

All this is illustrated in this figure.

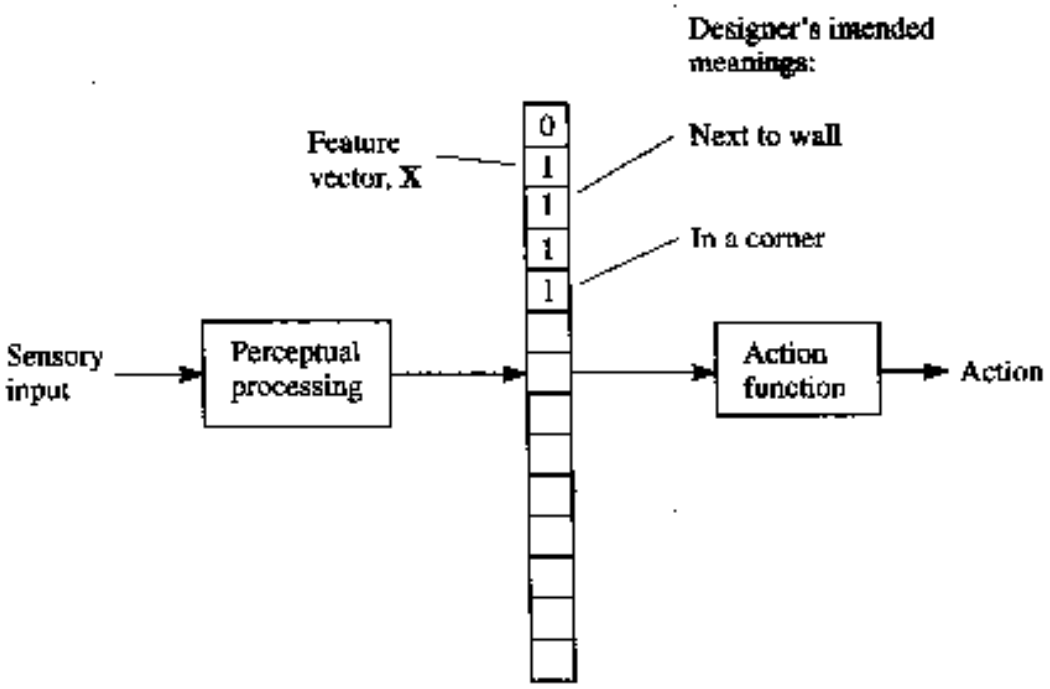


Figure 2.2

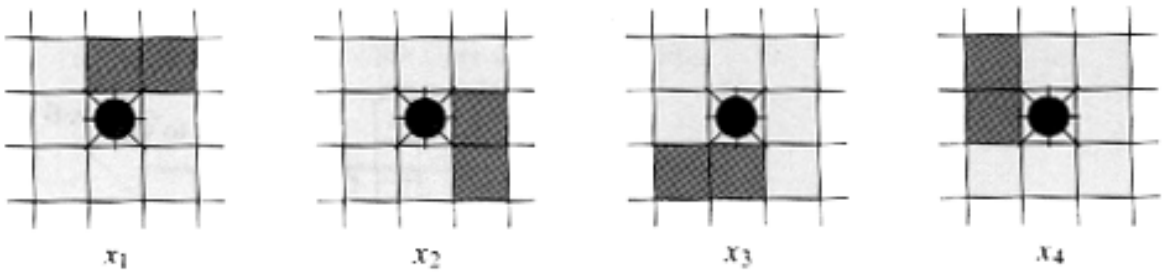
Perception and Action Components

This division into two phases is arbitrary. One criterion for the division is to put the mapping from common features to common actions into a reusable library.

Wall following robot

Percepts to features

It turns out that the feature vector needs only 4 components, x1 .. x4 which take boolean values. The values are illustrated in this diagram.



In each diagram, the indicated feature has value 1 if and only if at least one of the shaded cells is *not* free.

Figure 2.3

Features for Boundary Following

Recognizing these features in its environment allows the robot to achieve wall following behaviour.

Consider x1. x1 has value 1 if s2 = 1 and s3 = 0 or s2 = 0 and s3 = 1 or s2 = 1 and s3 = 1.

Features to Actions

Given the features, we need to map them to appropriate actions.The mapping is,

- If x1 = 1 and x2 = 0 then move east
- If x2 = 1 and x3 = 0 move south
- If x3 = 1 and x4 = 0 move west
- If x4 = 1 and x1 = 0 more north

Representing Action Functions

There are many ways of representing these action functions. Two very popular ways are:

- Production rules
- Neural networks

Here we just briefly consider production rules.

Representing actions with production rules.

Productions are ordered sets of rules of the form,

c1 --> a1

c2 --> a2

...

cn --> an

where the c's are conditions and the a's are actions. The c's are conjunctions which evaluate to true or false. The interpreter goes down the list in order to find the first rule who's condition is true, and then "fires" that rule's action part, in other words, executes that rule's action part.

Wall following robot example

The production rules from features to actions for the wall following robot example are

x4 & ~x1 -->north

x3 & ~x4 --> west

x2 & ~x3 --> south

x1 & ~x2 --> east

1 --> north

These rules cause the robot to go to one of the walls and then follow the walls for ever, either clockwise or counter-clockwise depending on its initial condition.

A variation might be a robot that goes to a corner and stays there. To implement this we would need a corner detecting feature, call it c. Then we could have production rules

c --> no-action

1 --> b-f

Where b-f refers to the boundary (i.e. wall) following procedure described by the five rules given earlier.

[top] [previous] [next] Questions?

A Reactive Agent with State

Imperfect perception

No agent ever perceives its environment perfectly. Random errors, or unavailability of information handicap the agent. To some extent, an agent can compensate for lack of perceptual information with memory. It is easier to find your way around a pitch black room if you have been there before. The wall following robot can illustrate the trade off between perceptual information and memory.

Impaired wall following robot.

Suppose the robot lost half its sensors. The corner ones are not there. Only north (s2), west (s4), south (s6), and east (s8) remain.

With these sensors alone the simple reactive agent cannot immediately recognize enough of its environment to find its way to wall following behaviour.

If, however, it can remember its previous feature vector, and remember its heading, then it can still achieve wall following behaviour even with its impaired "vision".

The wall following robot with state

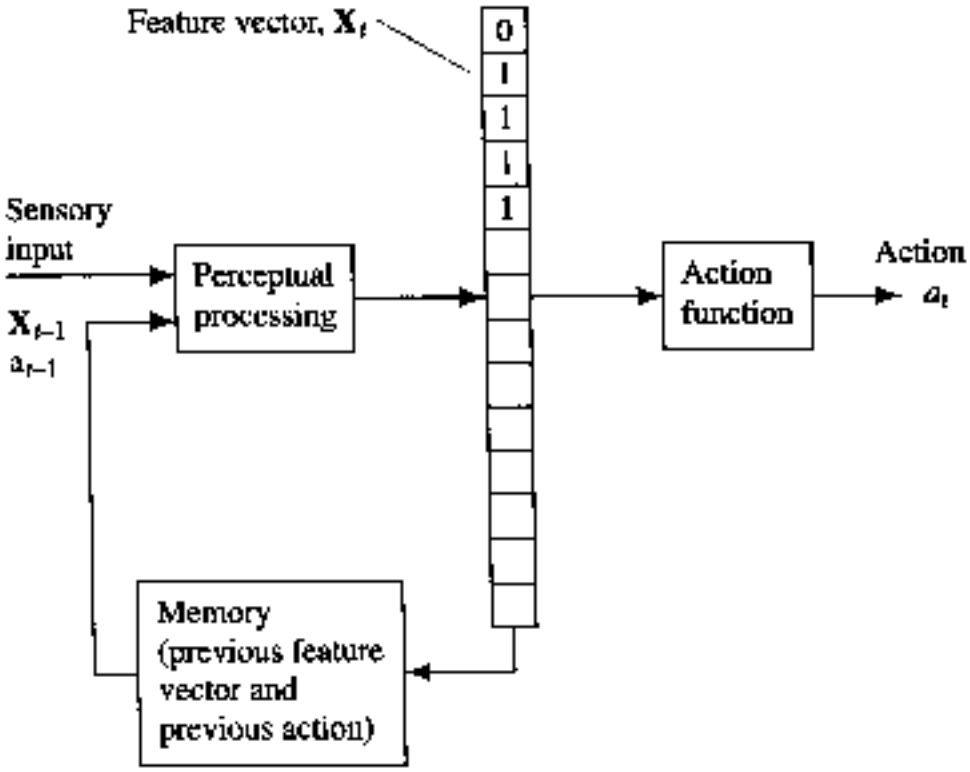


Figure 5.1

A State Machine

Here the environment is still represented by feature vectors but now the robot remembers its previous state, its previous feature vector and action, as well as using the features currently presented by its environment.

Calculating the features for the state based wall following robot

- For i = 2, 4, 6 , 8, wi = si
  These features are just the sensor inputs from north, east, south, west.
- For the absent sensory inputs s1,s3, s5, s7, we substitute four features, w1. w3. w5. w7., according to the following (clever!) rules:
- w1 has value 1 (true), if and only if, at the previous time step, w2 had value 1, and the robot had moved east.
- w3 has value 1, if and only if, at the previous time step, w4 had value , and the robot had moved south.

- similarly for w5 and w7.
- Otherwise the wi have value 0

The production system for the state based wall following robot

w2 & ~w4 --> east

w4 & ~w6 --> south

w6 & ~w8 --> west

w8 & ~w2 --> north

w1 --> north

w3 --> east

w5 --> south

w7 --> west

1 -> north

Role of memory

Note that the wall-following robot without the sensory impairment managed to achieve its behaviour without memory of previous inputs, features or actions.

If all of the important aspects of the environment can be sensed at the time the agent needs to know them, there is no reason to retain a model of the environment in memory. But sensory abilities are always limited in some way, and thus agents equipped with stored models of the environment will usually be able to perform tasks that memoryless agents cannot.

Agent Environments

Two approaches to representing environments for situated agents

- feature based representations
- iconic representations

Feature representation

The wall (boundary) following agents discussed previously use a feature based representation of their environments. Sensory inputs are converted directly to features which in turn are used to choose actions.

Iconic representation

The other popular approach is called by Nilsson, iconic representation. The name iconic implies some sort of picture, or model of the environment. The environment is simulated in the agent which then extracts features from the simulation. The simulation is updated as sensory information comes in.

A wall following robot using an iconic representation of its grid world:

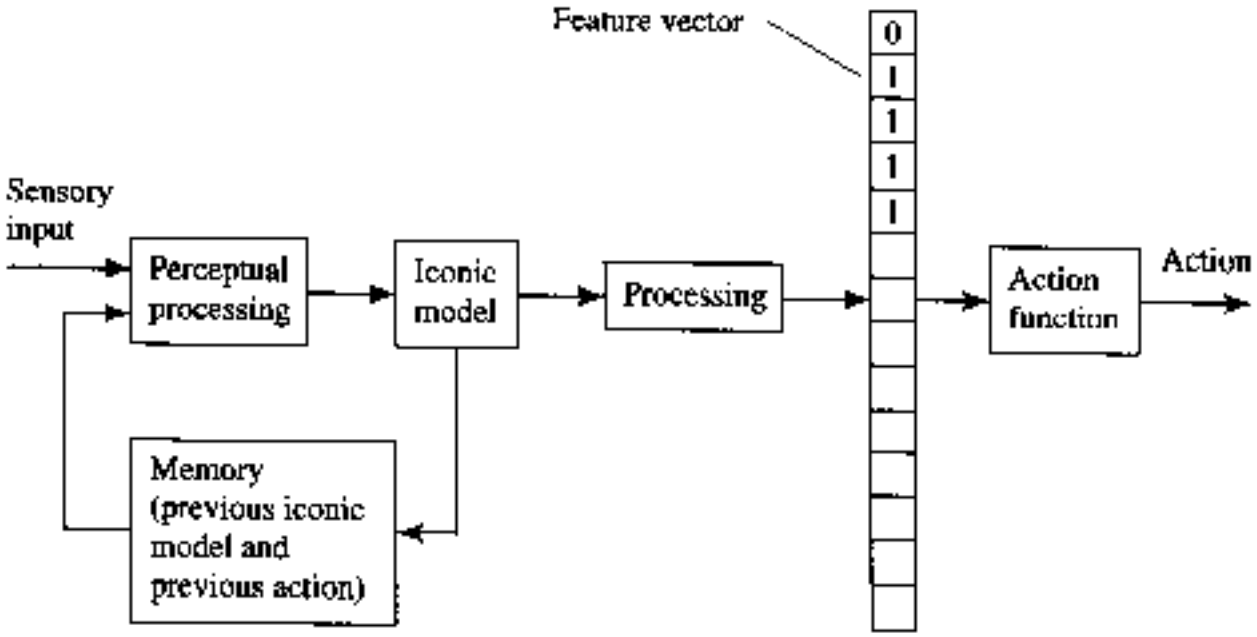


Figure 5.3

An Agent That Computes Features from an Iconic Representation

It might have a map like iconic representation like this:

	1	1	1	1	1	1	1	?
1	0	0	0	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	0	0	*R*	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	0	0	0	0	0	0	0	?
1	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?

Figure 5.4

A Maplike Iconic Representation

In the diagram, a 1 means unavailable (a wall), 0 means empty, ? means unknown. R is the robot, which should move west in this situation.

Or it could have a model based on potentials, like this:

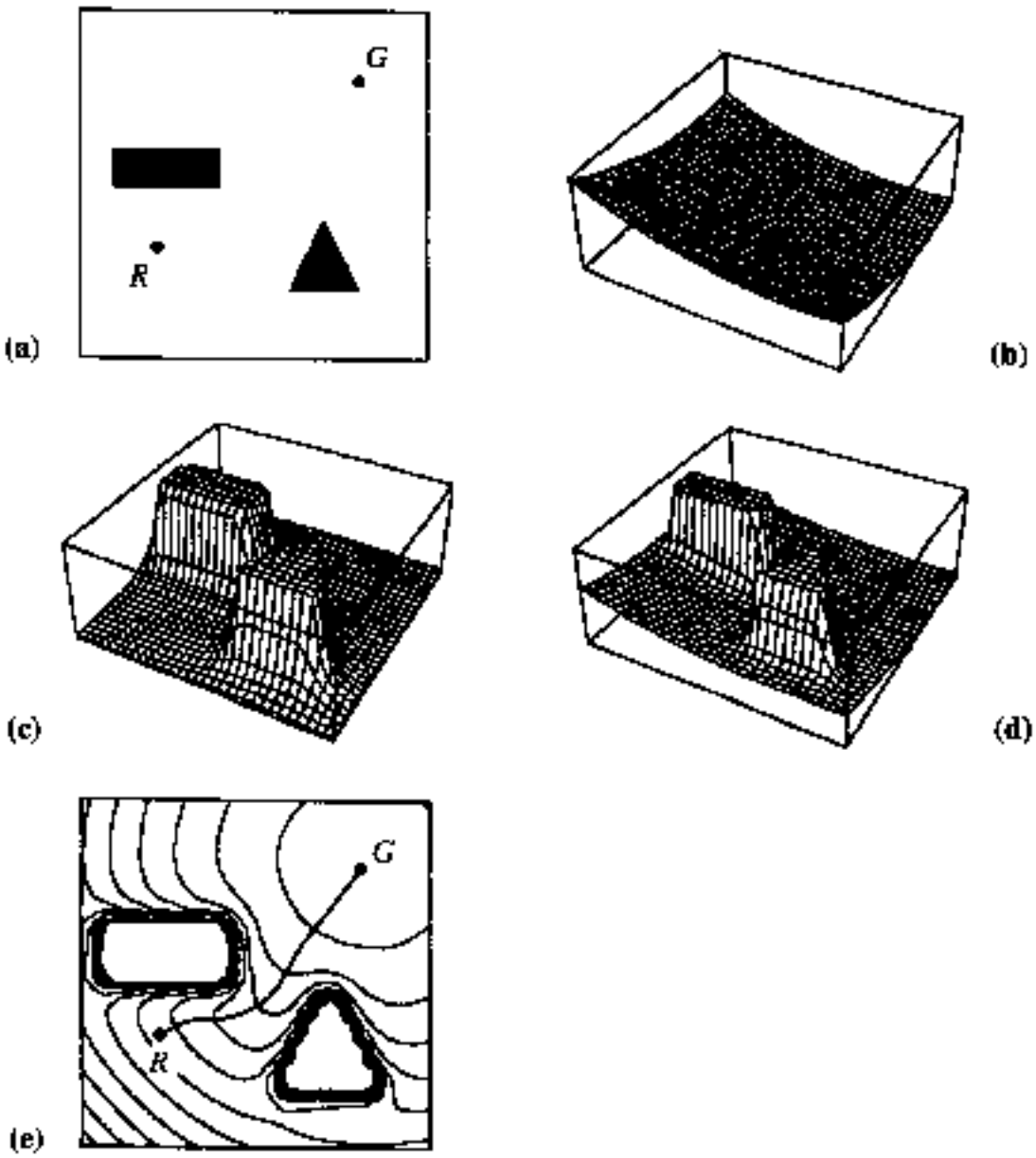


Figure 5.5

Equipotential Curves of an Artificial Potential Field (Adapted from [Latombe 199

The robot "slides" down the potential "hill" from where it is to the goal, G.

Toy Environments and Real Environments

The grid space environment is a "toy" environment in that it is very simple. Such

environments are useful to illustrate basic principles, and to do experiment in a controlled environment.

Questions have been raised as to the usefulness of toy environments. Do they "scale" well? That is, do the ideas and methods that work in toy environments retain their validity in real world environments? The jury is out on this question.

Ironically, it is the simpler reactive agents which have proved most useful in the real world.

Virtual Agents

Internet agents have attracted some attention recently. Their environments are servers. Compared to the real world where robots must exist, these virtual environments are simpler. But they are not toy environments. Such environments look promising as hosts for useful agents.

The rest of the course concerns mobile agents on the Internet.

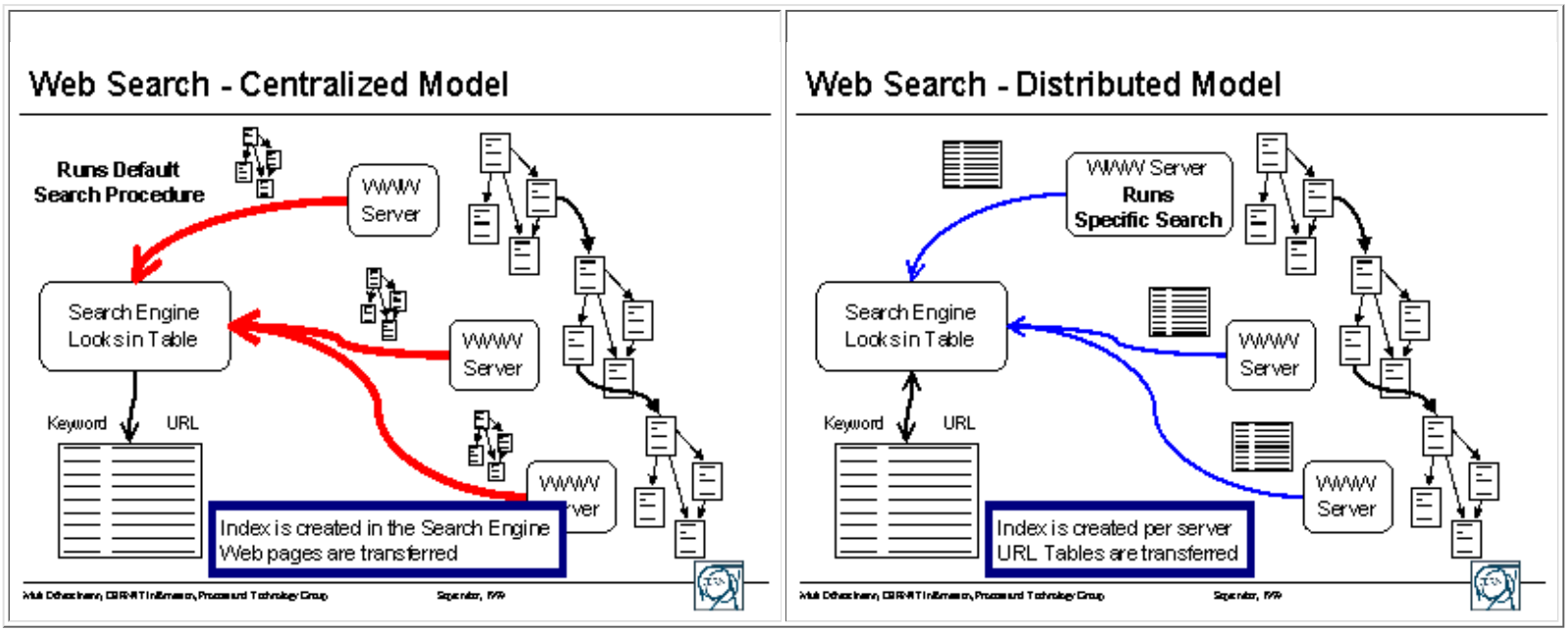
[top] [previous] [next] Questions?

CERN CSC99 Notes from Lecture 1

[by Mark Donszelmann]

The Agent Game

Example Agent Application -- Web Search



Centralized Model

In the centralized model, web pages (text) are brought down and searched to create a database indexing URLs with keywords. Yahoo, AltaVista keep these huge databases and are continually updating them. You can get your own such programs, such as WebFerrat (a free one) and build your own personalized web keyword/URL database.

Even thought these programs are called WebCrawler, WebSpider, WebFerrat, name which makes them sound like mobile agents, they are actually static. The pages (or at least their HEAD part) do the moving. The search engines are immobile.

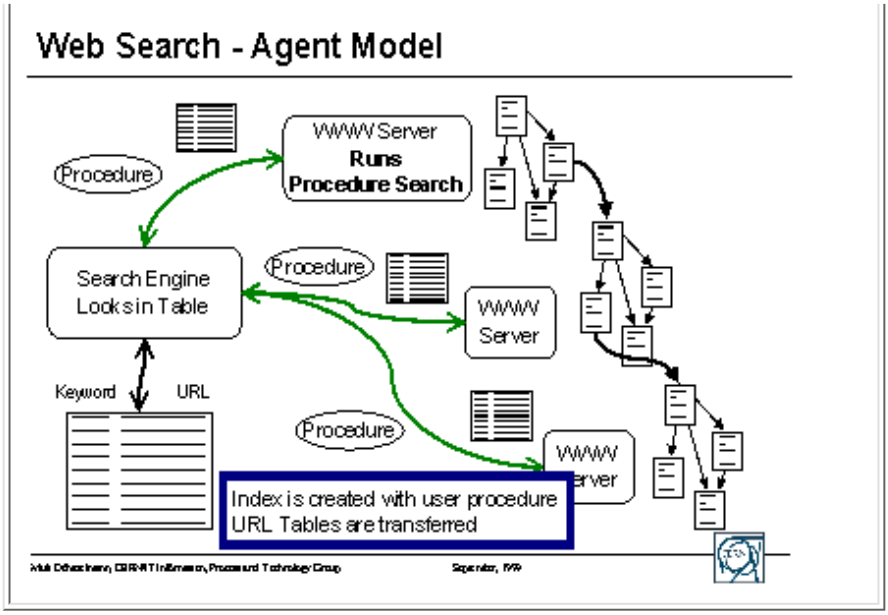
The result is lots of network traffic.

Distributed Model

This is also a static model. This time however, local databases of keyword/URL indices are created on each server which p rovides the corresponding web pages. Then the central database is made by merging the databases from all the servers.

The result is less network traffic.

The problem is coordinating all the local search engines. This is the "legacy problem".



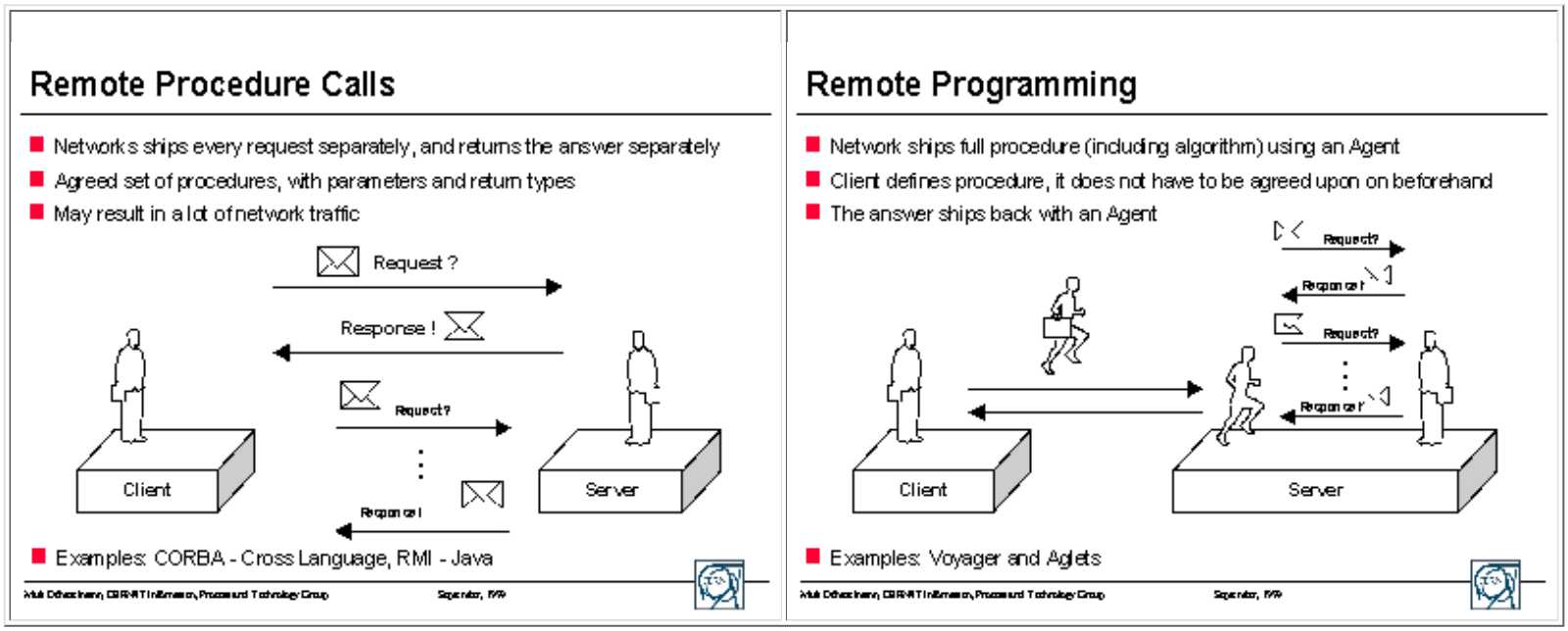
The Mobile Agent Model

The search engines (called Procedures in the diagram) move as mobile agents to the servers where the build local databases as in the previous case.

Moving the search engine actually adds a little extra network traffic as compared to the static distributed system described above. But the mobility solves the legacy problem. New, hopefully improved, search engines are easily installed.

Of course, the servers must be willing to accept agents!

Remote programming vs Remote Methods (RMI and RPC)



In the CORBA/RMI case the client knows methods or procedures on the server which can be invoked remotely. These procedures are pre-installed and are not changed dynamically at run time.

- Lots of network traffic generated by the calls
- The legacy problem again.

The remote program model helps solve both these problems.

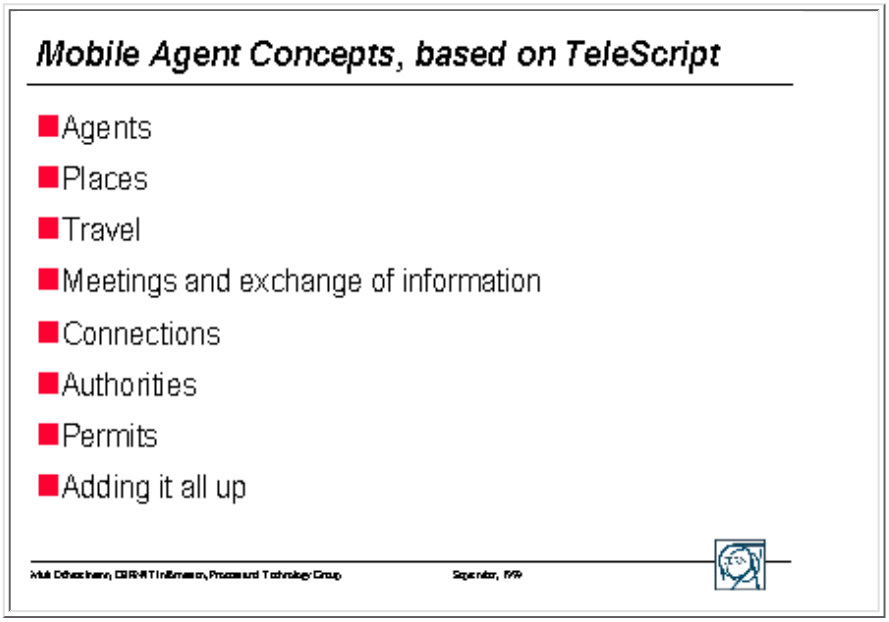
In addition, the agent essentially extends the power of the server dynamically because it can bring in additional functionality from the outside (subject to security controls). The agent arrives carrying an algorithm.

Advantages of Remote Programming over RPC

- Performance
  - Network carries fewer messages
  - The more work to be done on the server, the better
  - On slower networks the advantage is bigger
- Customization
  - The Client can extend (or invent) the functionality of the server
  - Ease of installation
    - no server installation
    - no agreed set of procedures
    - dynamic installation of the client's procedure on the server

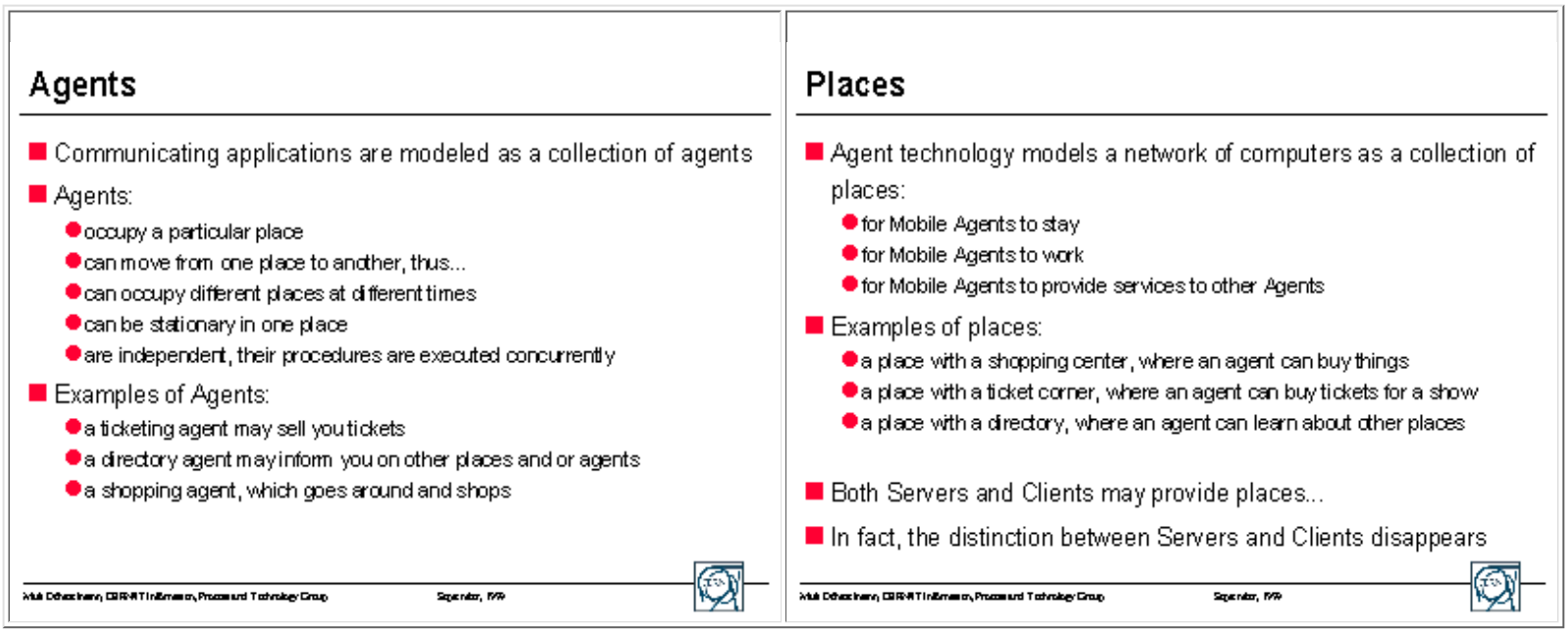
CERN CSC99 Agent Lecture 2 Notes

On Mobile Agent Concepts



These are very similar to those discussed in Lange & Oshima's book.

Agent and Place



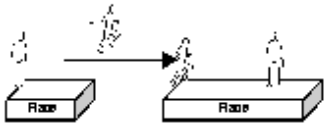
You can compare these points with Oshima & Lange's chapter 2. There are notes on this site based on the book.

- Some notes on agents
- Some notes on places

Travel and Communication

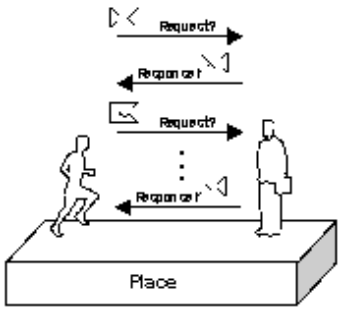
Travel

- Conventional programs can:
  - move to a place and then run
- Agents can:
  - run and while running, travel from place to place
  - for example, agents move to a place, obtain a service, and move back home
  - use a single "go" instruction to move themselves across the network to a different place
  - fail to travel if the place, where the agent is travelling to, is unavailable
- The language and underlying system permits this movement
- Example:
  - agents (such as in the Aglet system) can move while running
  - a C and C++ program cannot run and move while running, since neither its state nor its procedure are portable



Meetings and exchange of information

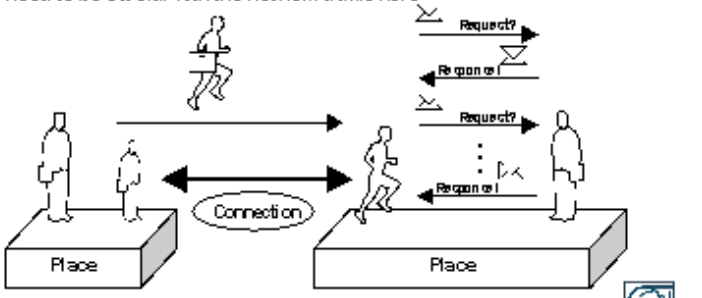
- Agents can meet if they are in the **same** place
- In a meeting they can call each others procedures (programmatic contact)
- Meetings motivate agents to travel
- Two agents may both travel to a place to meet
- Meeting is done by a "meet" instruction
- A meeting may fail, upon which the agent can decide what to do next



In his lecture, Donszelmann contrasts agents which travel from place to place with a batch program which goes to one place and runs only there.

Connections

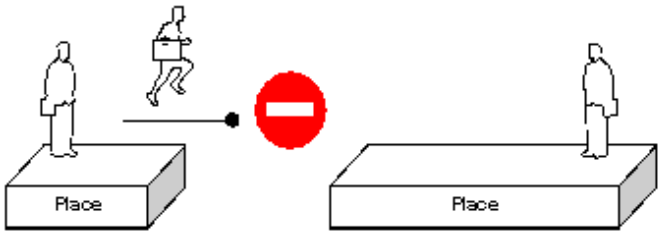
- Agents can connect to each other if they are in **different** places
- Example:
  - The User may want an update of what his Agent is doing
  - The Agent may need an update of what to do next...
- We need to be careful with the network traffic here



Authorities and Permits

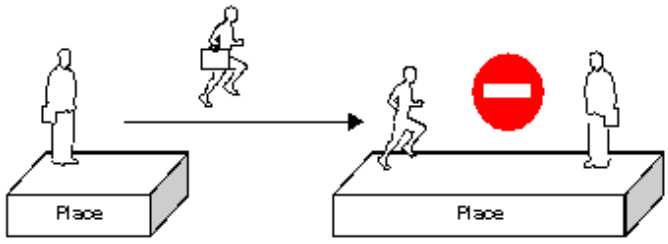
Authorities

- Places and Agents should be able to:
  - Find out who an agent is
  - Make sure he is who he says he is
  - Find out where the agent is from
  - Make sure he is from the place he says he is from
- A place may deny entrance of an agent which is not authorized



Permits

- Authorities limit what agents and places can do by assignment of permits
- Example:
  - Execute: An Agent can be permitted to create other agent (with same or fewer permits)
  - Use: An Agent can be permitted to make use of some (local or remote) resource
- Permits are important, since agents execute (unattended) in server machines



Summary

Adding it all up

- The power of agents is to provide higher-level composite services from a set of basic (distributed) services, by having agents travelling to several places
- Example:
  - An agent travels to a ticketing place to buy tickets for a theatre show
  - Afterwards it may stop by the taxi place to book a taxi for that evening
  - **Note**: the agents decision to book a taxi for a certain time depends on the availability of the tickets for the theatre
- Places (*on computers*), Agents (*mobile programs*) and Travelling (*using the network*) are used as a big **platform** to put all of this in motion

The Mobile Agent

A mobile agent has **five properties**

1. **State**: needed for the agent to resume computation after traveling.
2. **Implementation**: needed for location-independent agent execution.
3. **Interface**: needed for agent communication.
4. **Identifier**: needed to recognize and locate traveling agents.
5. **Principals**: needed to determine legal and moral responsibility.

State

The agent must carry sufficient state information with it to resume execution when it moves from one host to another.

An agent's state is a snapshot of its execution.

Two parts to state

- **Execution state**. Includes the program counter and the execution stack.
- **Object state.** values of instance variables of the agent object.

Java does not allow access to the execution stack so when an Java agent such as an Aglet moves from one host to another, it loses its execution state. (Even if it could keep this information, it would prove useless if the next host was a different type of platform from the host just vacated.)

However, usually the information needed to resume the computation can be kept in the object's instance variables (for example, partial or intermediate results of a calculation). The saved information can be used as initial values in the resumed calculation.

Implementation

Of course an agent, being a computer program, needs code.

- A mobile agent can get its code in 3 ways,
- Take the code with it
- Hope the code it needs is already at its destination
- Get the code from some other location

Interface

The agent's interface allows other agents and systems to interact with it.

Such an interface could provide

- A set of public methods that other agents can invoke
- A full fledged messaging interface which allows agents to communicate with some sort of "agentspeak" such as KQML (Knowledge Query and Manipulation Language). KQML derives from AI research on speech act theory and provides a rich communication language based on "performatives".

Identifier

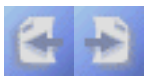
A unique name which is unchangeable throughout its lifetime.

Principals

These are the humans responsible for the agent's actions. Oshima & Lange suggest a division of responsibilities.

- Manufacturer. The author of the agent.
- Owner. This person has the moral and legal responsibility for the behaviour of the agent.

See also the CERN course notes on agents.



Places

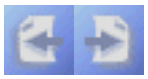
Lange and Oshima call the environment provided by the network hosts for the agent to operate in, a place. In the Aglet world, a place is called a context. Places should be safe places both for the agents and their hosts!

The place is a kind of operating system for the agent.

There are at least 4 aspects to a place

- **Engine**. the "workhorse" and virtual machine for one or more places. For aglets, this is the JVM.
- **Resources.** databases, processors, and other services provided (or not!) by the host.
- **Location**. The network address of a the place. The IP of the host, and a port on which the place awaits agents.
- **Principals**. those legally responsible for the operation of the place. As for the agent, there are two principals.

See also the CERN course notes on places.



The Internet Agent and its Environment

Situated agents such as robots inhabit environments which are spatial (Euclidean 3-space) and in which the laws of physics apply. Purely communicative (Internet) agents also inhabit an environment of sorts. So alien is this environment to us that Jacques Ferber in his boot Multi-Agent systems goes so far as to say that purely communicative agents have no environment in the normal sense of the word.

Nonetheless the word 'environment' is commonly used to refer to the places 'inhabited' by Internet agents. These places are servers of one kind or another.

Here is a summary of Lange and Oshima's point of view.

The agent

This is a piece of software which has its own thread of execution. Having a separate thread gives it a certain autonomy from its operating environment. A mobile agent is able to "close up shop" on one host, and pack itself off to another.

The environment

On the Net an agent's environment consists of a server program running in a process,

This server provides a "sandbox" which secures the host computer from possibly malicious visiting agents.

On the positive side, the environment can allow the visiting agent to access the host in a controlled way. Therefore the agent's environment usually consists of the server plus certain parts of the host system.

"Sensors"

The Internet agent's "sensors" are just methods which it can call to read information from the server or the host.

"Effectors"

Similarly the "effectors" of the internet agent are also methods which write to the environment, or cause methods of the environment to be executed. The agent normally interacts with its server, but with permission can also interact with the host file system, or database, for example.

Note that the concepts of effectors and sensors are rather artificial when applied to Internet agents. They really belong in the world of situated agents.

Inter-agent communication

In addition to sensing/effecting their environments, most Internet agents also have some means of communication among one another. Often, a group of agents collect at one server and interact by sending each other messages, either directly, of via the environment provided by the server.

Of course, it is also possible for the agents to stay at their home location and send messages to one another over the Net.

[top] [previous] [next] Questions?

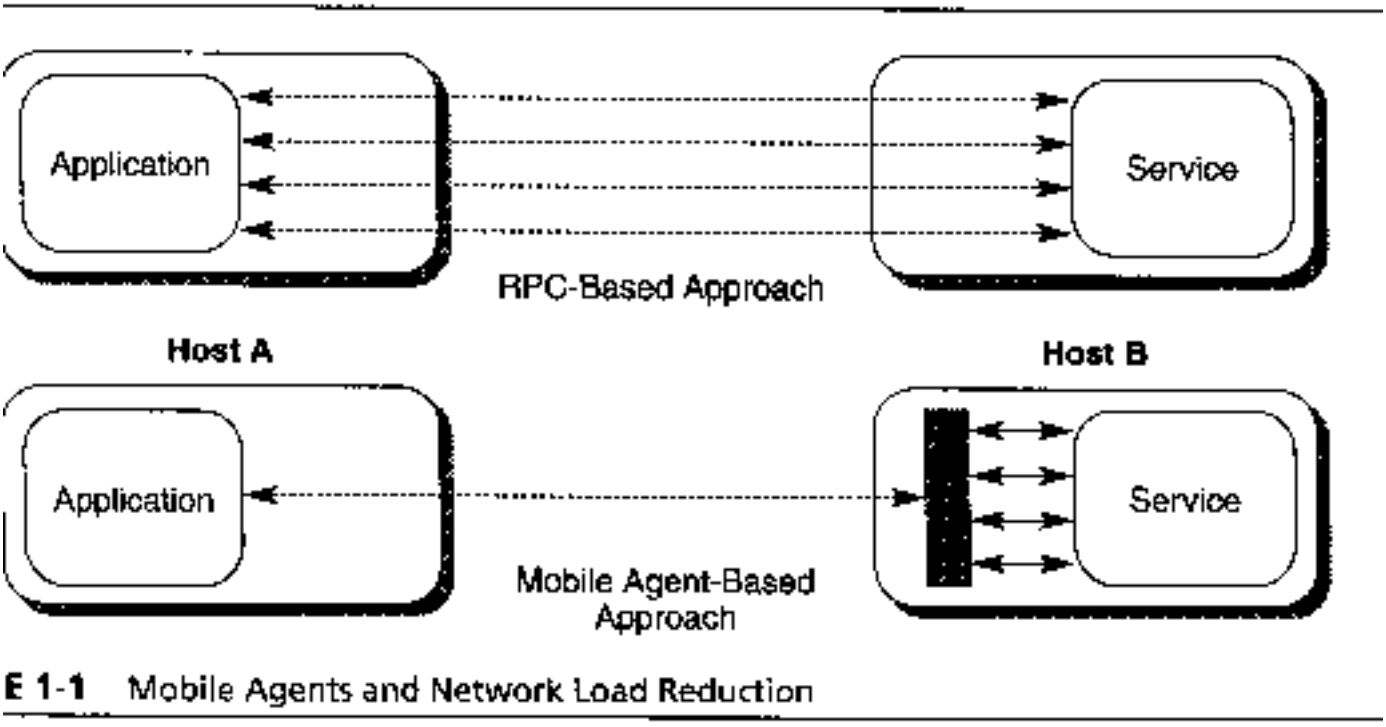
Agent Mobility

When one talks of Agents on the Net, one quite often means mobile agents. Mobility confers a number of possible advantages, according to Lange & Oshima (p. 3).

Seven Advantages of agent mobility.

(Or are some of these also the advantages of Object mobility? For a brief discussion of the distinction between mobile agents and mobile agents, click here.)

1. They reduce network load



E 1-1 Mobile Agents and Network Load Reduction

The moral is, sometimes it is better to bring the computation to the data, than to bring the data to the computation.

An example: Weather forecasting calculations

A server has a database with gigabytes of real time data. Other people at other locations might want to do analysis of the data. To copy the data to their sites would take a lot of time and bandwidth. It would be better to encapsulate the required calculations in an agent, send it to the original site, do the calculations there, and return the results.

Another example: Searching for a particular document on the Web

Assume there is some criterion for success, not just the title, possibly a string, or several strings. Suppose you know it is on one of 1000 web sites, some of which have gigabytes of documents.

You could do the following,

```
For each web site
    For each document
        Download the document
        Search document using the criterion
        If found, stop and inform the user.
```

This method could take "forever", and clog the Net with hordes of up-needed documents.

With a mobile agent system you could do the following instead:

```
Clone 1000 identical search agents
For each agent
    Move each agent to a target site
    For each document on each target site
        Search document using the criterion
        If found, send document to user and self destruct
```

Send messages to other clones ordering them to self-destruct

Note that this method exploits the potential parallelism in mobile agent system

2. They overcome network latency

Consider some sort of large control system, controlling manufacturing robots in a large factory. If these need to respond in real time to changes in their environment. If the controlling programs are centralized, network traffic may slow responses in an unacceptable way. Dispatching agents to control the robot processes locally can solve this problem.

3. They encapsulate protocols

In the usual system, each host owns the code which interprets the incoming and outgoing data. Updating these on large systems can be a problem. If the protocols are encapsulated in agents, new versions can be sent to all the hosts at the same time.

4. They execute asynchronously and autonomously

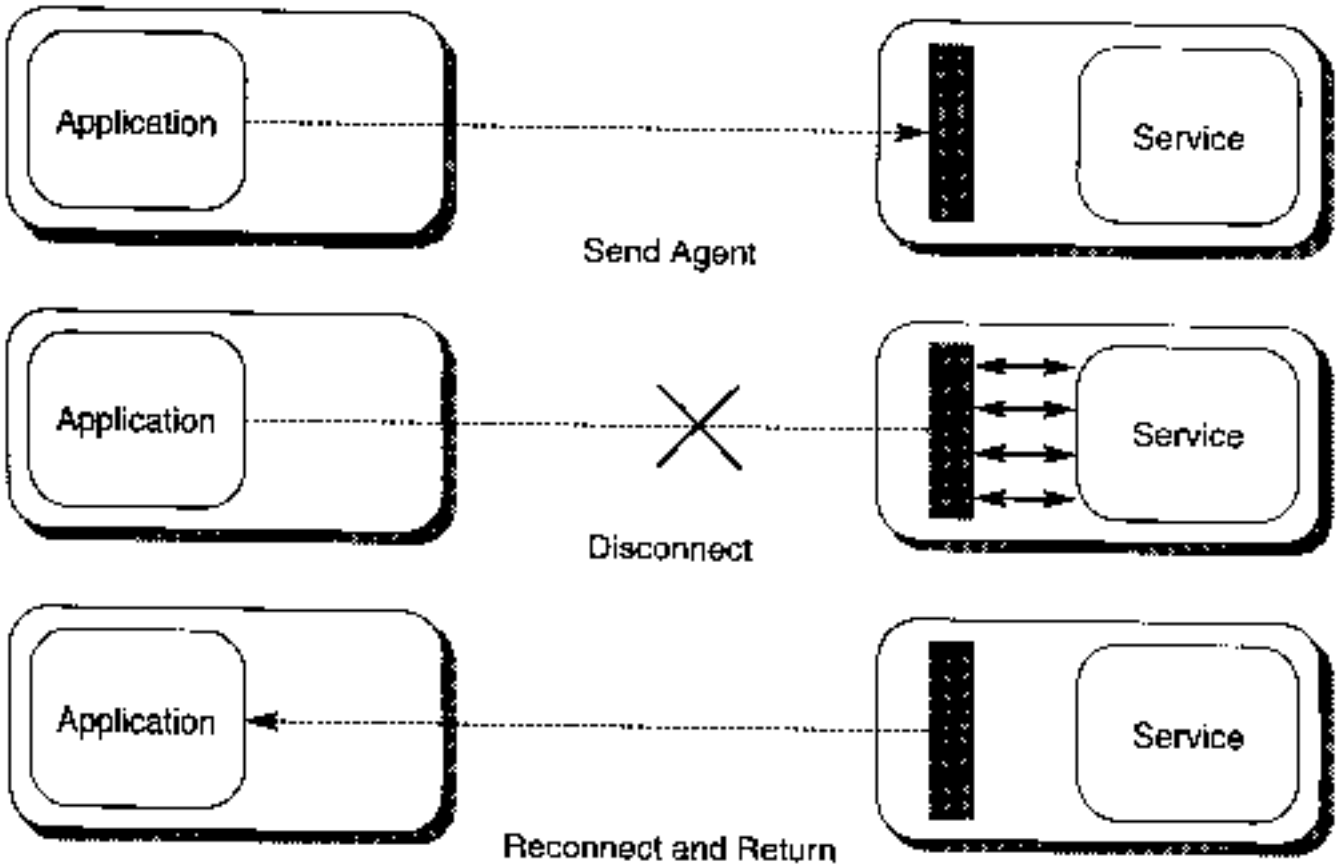


FIGURE 1-2 Mobile Agents and Disconnected Operation

With the rise of mobile computing, the problem of disconnects, either by choice, or accident, becomes increasingly important. An agent sent from a mobile computer is autonomous. The sender can disconnect. The agent does its work at a remote host, and then waits. Whenever the mobile user is ready, she reconnects and retracts the agent with its result. There need be no synchronization between the connection and the computation.

5. They adapt dynamically

Agents, with a little intelligence built-in, can adapt to changes in their execution environment. For example, if the host signals shutdown, the agent can pick up and go to another host to continue its work. Groups of agents can distribute themselves among hosts to achieve maximum efficiency.

6. They are naturally heterogeneous

They are usually transport layer and host type independent. For example, many are written in Java. Agents are dependent only on the execution environment provided by their hosting server (e.g.Tahiti for Aglets). There independence of hardware and communication mechanism makes them suitable as a "glue" for system integration of heterogeneous systems.

7. They are robust and fault-tolerant.

This is because of their mobility. If something is going wrong at one location, they have a chance to escape and continue at another.

[top] [previous] [next] Questions?

Mobile Objects vs Mobile Agents

In his book, Programming Mobile Objects in Java, Jeff Nelson contrasts mobile objects with mobile agents (p. 70-77). For him, the main distinction concerns autonomy. Nelson's mobile objects have no autonomy. They are slaves of the programmer. Mobile agents, on the other hand, have some degree of autonomy.

Earlier we related autonomy to intelligence. So the difference might relate to how much AI is in the agent/object system.

The difference between mobile agents and mobile objects is fuzzy. In cps720, the aglets discussed are neither very autonomous nor very bright! They could just as well be classed as mobile objects.

A practical difference

As Jeff Nelson points out, the autonomy of agents comes at a price. The agents must be provided with a "playpen", the "sandbox", at each host. That is, each host must have a server to provide an environemnt for the agents.

Furthermore, an API must provided to go with the agent and its environment. The agent is only as powerful as the API provides sufficient methods. In other words, the agent's "effectors" and "sensors" are predefined in the API. This reduces flexibility.

Mobile objects are just objects. Their use requires no special servers or API. They have no autonomous behaviour at all. They are just building blocks, along with ordinary objects of object oriented distributed systems.

Nelson asks what use agents would be in building a complex point of sale internet site.

Comment

- While not a point of sale internet site, the case study discussed in Todd Papiaoannou' s PHD Thesis shows a complex set of business rules implemented in an agent system using Aglets.
- Jeff Nelson's mobile objects are sometimes refrerred to as "code on demand". Code on demand is certainly a useful paradigm. Applets are an example.
- A deeper discussion is in chapter 2 and 3 of Papiaoannou where he compares mobile agents with other forms of distributed computing, basing his discussion on the perceived importance or unimportance of location. A summary discussion of this point is in the next section of these notes.

[top] [previous] [next] Questions?

Network Computing Paradigms

Where do mobile agents fit into network computing? Lange and Oshima suggest the following. paradigms, from the perspective of the software developer.

- Client-Server
- Code on demand
- Mobile Agents

These diagrams refer to "know-how" and "resources:. Know-how is the code that enables the services provided by the system, i.e., access to the resources. This is the knowledge in the system. The resource in question is often a database of some sort which supports the services.

Client-Server Paradigm

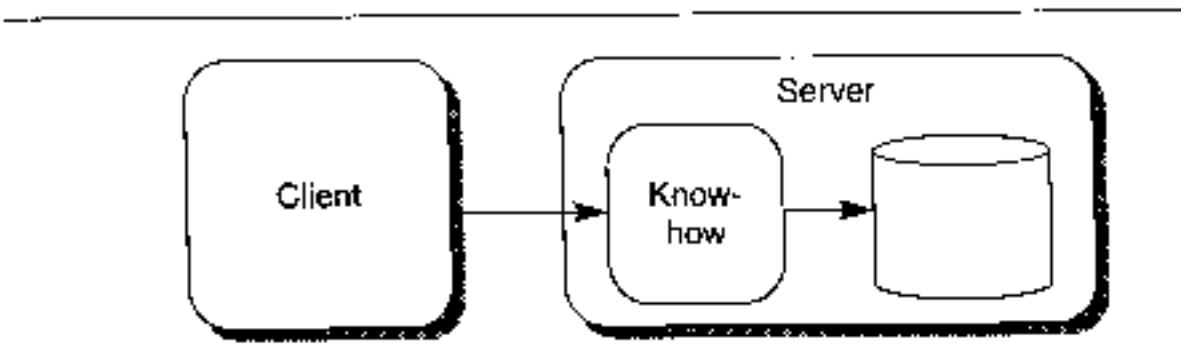


FIGURE 1-3 Client-Server Paradigm

This of course is the best known and most common paradigm.

In this case, the server holds both the resources and the know-how, that is, the processes that enable the services. The server has it all. An example would be the Common Gateway Interface (CGI) on a Web server. All the power is on the

server side. The client usually just has a web browser which displays HTML forms to be filled out by the user.

Code on Demand Paradigm

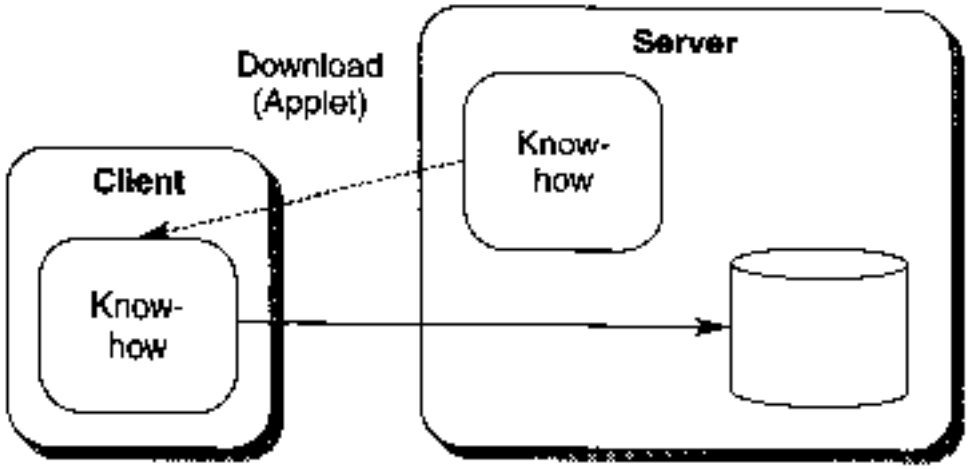


FIGURE 1-4 Code-on-Demand Paradigm

This model gives more power to the client side. Java Applets are the classic example of this paradigm. Active code is downloaded to the client. Consequently, the client shares some of the know-how. Knowledge is more distributed than in the original client-server model.

Code on demand has developed far beyond Applets. One hears of ASP's, Application Service Providers, as well as ISP's. An interesting implementaion of this technology with Java is Sun's Java Web Start.

Mobile Agent Paradigm

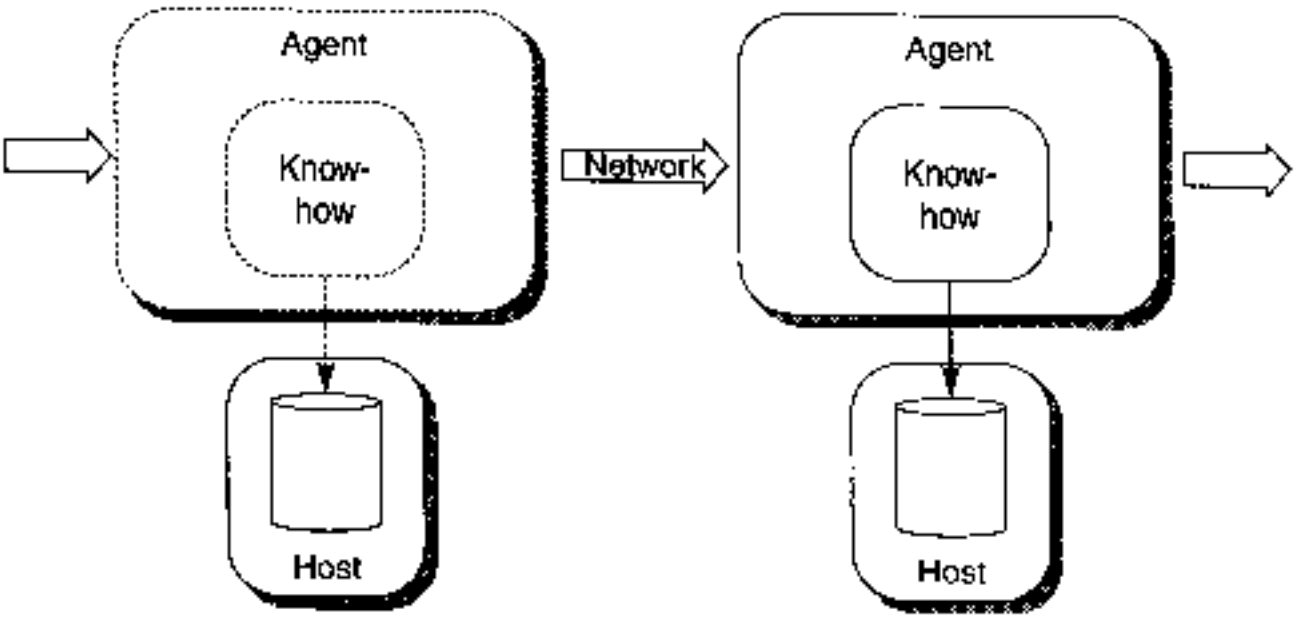


FIGURE 1-5 Mobile Agent Paradigm

Here the watchword is flexibility. Any host on the network can have any combination of know-how, resources and processing ability. An Agent based network is a kind of peer to peer network.

[top] [previous] [next] Questions?

The Question of Location Transparency in Distributed Systems

This page discusses the desirability, or not, of location transparency in distributed systems. A much more detailed discussion can be found in Chapters 2 and 3 of Todd Papaioannou's Ph.D. Thesis.

Location Transparency

What is location transparency. In a distributed system it is the idea that the resources accessed by a user can be anywhere on the network without the user having any idea where the resource is located. A file could be on the user's own PC, or thousands of miles away on another server, for example. The user would access it in the same way. Sun Microsystems slogan, "The network is the computer" sums this idea up. Many modern distributed systems illustrate location transparency. Consider, for examples, Java RMI, CORBA, or Microsoft's DCOM.

Remote Procedure Calls (RPC)

In a UNIX system, on a single machine with Von Neumann architecture, programs run in separate processes and inter process communication (IPC) is implemented via pipes, files, and shared memory. Remote Procedure Calls were developed in the early 80s to extend inter process communication to remote computers accessed across a network.

Java, being object oriented, implements RPC in its own way, called RMI. The basic idea is the same.

The goal of RMI and RPC is to make the calls transparent to the programmer and the user.

With RMI, for example, once so called stubs and skeletons are set up, and a name server invoked, the RMI programs are implemented in much the same way as a program confined to a local machine would be. The situation is quite similar with CORBA.

IPC: see *Structuring Distributed Systems* (39)

In systems such as RMI and CORBA, location, while not totally out of the picture, is not a central concept to these versions of the distributed computing paradigm.

RPC: see *Structuring Distributed Systems* (42)

Most present day distributed systems are based on RPC, RMI, CORBA or something similar (such as proprietary systems for Microsoft or Novell). Are such systems the only answer to the problem of implementing distributed systems. In Papaioannou's words ,

"By following the location transparency abstraction, contemporary distribution infrastructures in effect attempt to provide a virtual von Neumann machine. That is, by trying to fool every component in the system that they exist within the same address space, the overall effect is the creation of a

virtual machine." [33/44] See diagram on p. 34 (46).

Or do they have problems, not just implementation problems, but fundamental architectural problems. Todd Papaioannou thinks they do.

Problem with Location Transparent Systems based on RPC

In a phrase, they don't scale well. Such systems eat bandwidth because of the number of messages (for example RMI objects for return values, method parameters etc) which must be exchanged. As the number of servers increases, the number of messages increases exponentially. The system slows down.

Another problem is that these systems assume a very high reliability on the part of the network system.

See table from *Structuring Distributed Systems*. [37 or 49]

So the lovely idea of location transparency has its dark side.

In the mid 90's some began to believe that distributed systems are intrinsically different from local systems and should be treated differently.

Mobile Agents - a solution?

Mobile agents represent an approach to distributed computing which is opposite to RPC with respect to location transparency. Agents always know where they are, and must know the location of some of the other agents in this system, location, or place, is a central concept in the mobile agent paradigm.

An Architectural Mismatch

The question arises, why might making location important in the design of distributed systems be better than hiding location, making it appear invisible, transparent? Todd Papaioannou offers the following explanation.

The mobile agent paradigm brings us closer to the successful IPC abstraction. IPC is well matched to the Von Neumann machine architecture using, as it does, shared memory and files for (local) communication. Mobile agents emphasize, as much as possible in a distributed system, *local* interactions. Once settled in on a server, the mobile agent can take advantage of IPC.

[See diagram on p. 51 (63) *Structuring Distributed Systems*]

Of course the mobile code still must move, at least once in a while, and must also send messages across the net. With these activities mobile agents run into some of the same problems as RPC based distributed systems. But the problems are reduced since network traffic is usually reduced. And the agent (or its programmer) has more choice, choice on whether to stay put, or move. In principle this decision can be based on dynamic local and network conditions, and on the resources available at servers.

But to do all this, the agent has to know where it is and where it can go to. In short, it needs to know about location explicitly. This is just the opposite of current distributed systems.

Whether Todd Papaioannou is right to say that mobile agents have significant architectural as advantages over current "immobile agents" remains to be seen.

[top] [previous] [next] Questions?

What are aglets?

Aglets are mobile internet agents. They are implemented in Java. They were developed by IBM Japan. Environments are provided on hosts by specialized servers which understand the aglet transfer protocol (ATP) and provide security and other services. The Aglet distribution is provided with such a server, called Tahiti.

The design of aglets is modeled on that of Java applets. The word 'aglet' is a contraction of 'agent' and 'applet'.

Getting Started with Aglets

This is a short tutorial to help you to get your aglets up and running. We use two Aglet demo programs, HelloAglet and CirculateAglet which come with the Aglet distribution. In August 2000 IBM released the Aglets system as open source. You can find Aglets at SourceForge. The Open Source folks have ported Aglets to the Java 2 system (e.g. JDK 1.3) which involved working with the Java 2 security system. They have packaged the system veryconveniently if you want to install it on your own system. (See notes below). This version is Aglets 2.0.

On Solaris at Ryerson SCS

These notes are specific to the host, jupiter.scs.ryerson.ca, run under the auspices of the School of Computer Science at Ryerson University. The Aglet files have already been unziped and you need only do a few things to get Aglets working.

File Arrangement on jupiter

You will find the necessary Aglet files in the directory /software/aglets and its subdirectores on jupiter. Most of the Aglet system istself is in a collection of jar files in the lib subdirectory.

The directory, aglets, contains fthe files you need to put on your own account. These files are

- **tahiti** (called agletsd in the original distribution) A script for running the Aglet server
- **aglets.props** A configuration file for the Aglet server.
- **.java.policy** Configures Java 2 security
- **.keystore** Contains encryption keys

(use ls -la to see the last two.)

Files on your account

Create a directory named aglets, with a subdirectory, public. Copy the files, tahiti and aglets.props to your aglets directory.

Copy .keystore and .java.prolicy to your home directory.

Go to your public directory and makes some links

- ln -s /software/aglets/public/examples
- ln -s /software/aglets/public/com

(The second link allows the use of IBM extensions in the com.ibm.agletx directory - see below.)

Two changes in aglets.props

You need to make two changes in the file aglets.props.

Find the line aglet.class.path=/software/aglets/public, and add : followed by the path to your public directory.

Find the line aglet.public.root, and change it to point to your public directory.

Fixed and mobile code and the CLASSPATH

Aglets are mobile agents. In other words, they are programs whose code is mobile. It can move from one execution environment to another.

Much Java code is not allowed to be mobile. Any Java code on the CLASSPATH cannot be mobile. The AGLET_EXPORT_PATH environment variable can be used to point to classes which can be mobile, i.e., can be Aglet code.

A problem can occur if you are developing an Aglet in a directory which is on the classpath, for example, the current direcotry if '.' is on your classpath. After compilation your Aglet class file is on the classpatha and therefore cannot move to another system. One way to avoid this problem is to write a one line script beginning with javac -cp <set appropriate class path here> ..... instead of setting the CLASSPATH environment variable permanently.

You also need the aglets jar file on your classpath to compile Aglets.

Examples

- *On MS Windows*: javac -cp .;c:\aglets\lib\aglets-2.0b0.jar mya1*.java
- *On jupiter:* javac -cp .:/software/aglets/lib/aglets-2.0b0.jar mya1/*.java

These examples assume you are in the public subdirectory of the aglets directory and that your code is in the directory mya1, a subdirectory of public. It is further assumed, in this example, that your Aglet code is in a package named mya1.

The com.ibm.agletx Package

If you examine the directory structure of the Aglet installation you will see a series of directories /software/aglets/public/com/ibm/agletx/patterns and /software/aglets/public/com/ibm/agletx/util.. These are useful classes which you may want to use in your Aglets. Some are used in the itinerary example below.

The agletx package ('x' for exportable) are kept separately, and NOT put on the class path, because these classes could travel with mobile aglets. For security reasons, classes on the Java class path cannot move. Be careful in you own programs not to put mobile code on the class path. (You may have to do this for compiling, but not for execution).

Invoking Tahiti on jupiter

All Aglets need an environment, or context, to support their execution (and prevent malicious visiting aglets from damaging a host). The Aglet distribution provides a program called Tahiti which provides this context, and a convenient user interface. Tahiti is configured using a property file such as the one supplied in /software/aglets.

If you have set up as described above,then do the following to run Tahiti.

Go to the aglets directory and type,

tahiti -f aglets.props -port 12000

(12000 is just an example. You will probably be assigned some port numbers.)

This command should bring up the Tahiti server viewer.

You can also run without a GUI (useful with telnet):

tahiti -f aglets.props -port 12000 -nogui

As a preliminary test of your setup try this.

In the Tahiti viewer, click the Create button. In a popup window, a list of available Aglets (supplied with the distribution) should appear. Choose, examples.simple.DisplayAglet, and click Create. A message should appear in the Tahiti window.

Aglets On NT or Win95/98,ME,2000

The Aglets 2.0 release has some features which make setting up the Aglet Server (Tahiti) much easier than before. You download the version 2.0 zip file from SourceForge and unzip it.

Bundled with Aglets is Apache's ant program. This is a Java replacement of a makefile. To configure the installation, just go to the bin directory and type, *ant*. Among other things, ant creates the security files .java.policy and .keystore. You can also use ant to put these in the right place. Just type *ant install-home*.

To bring up the Tahiti server with the configuration prodcued by ant, type (in the bin directory),

agletsd -f ..\cnf\aglets.props

The server will run on default port 4434.

Running some examples

NOTE: In these examples, various port numbers are used. If you try these examples, please use your assigned port numbers to avoid clashes with other students' servers.

Also note that you *may* need to modify the aglets.props path settings for some of these examples.

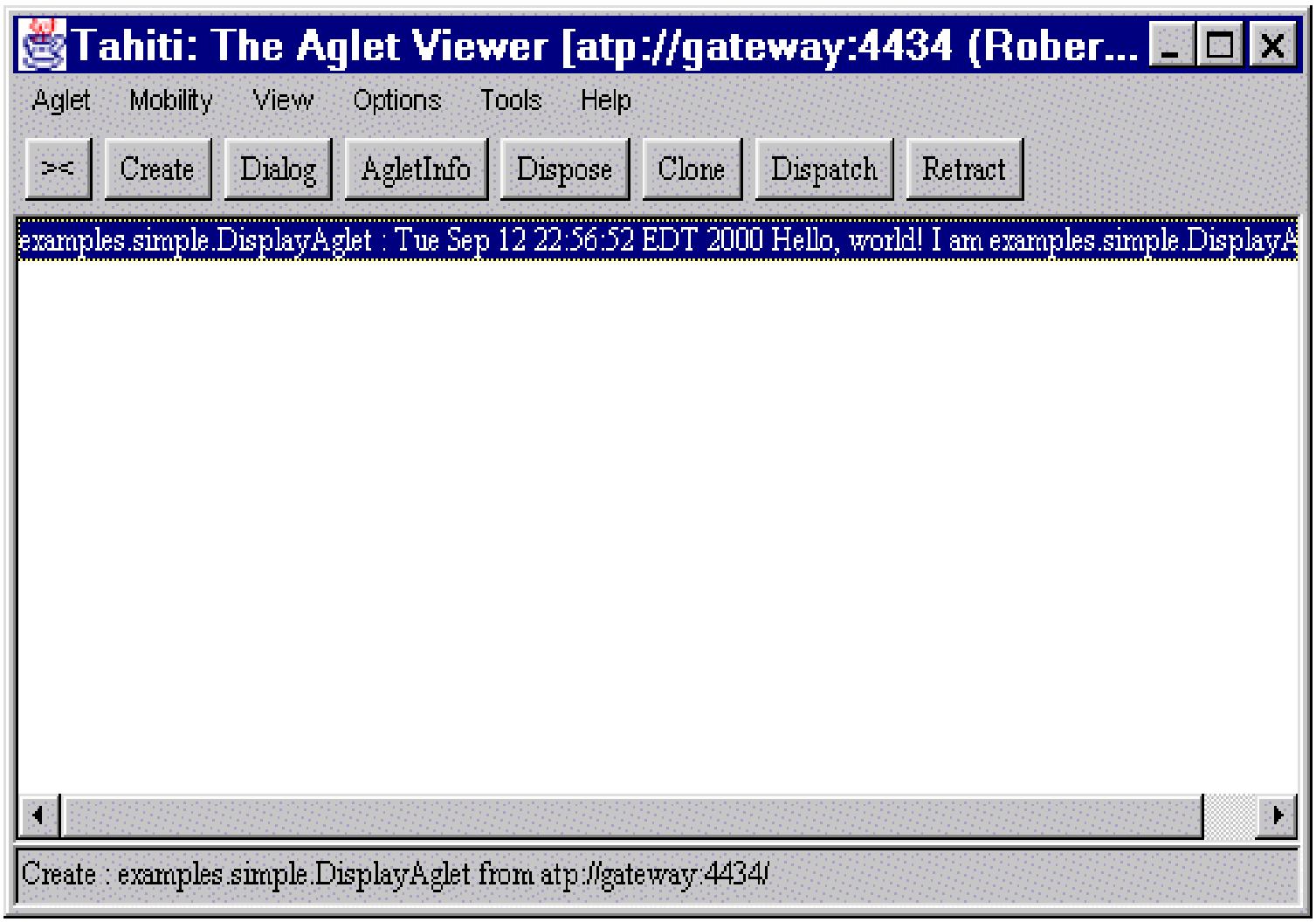
Example 1.

The simplest example is *examples.simple.DisplayAglet*. It is useful to check if the Aglet system is set up correctly. You load ithe Aglet into Tahiti using the Create button. Notice the fully qualified names for Aglets.

Set up another Tahiti (or have a friend do so) and use the Dispatch button to send the Aglet to the other Tahiti server. In the list displayed by the Dispatch button, enter the URL of the other server. (Don't forget that the protocol is atp, not http!)

The Aglet should disappear from your Tahiti and turn up at the other one.

Then use the Retract button to bring your Aglet back home.



Example 2. HelloAglet

The HelloAglet is dispatched from one Tahiti server to another where it says "Hello, world", and then returns to its origin to be disposed of.

1. Assuming you have set up your directory structure as above, make a hello subdirectory in the examples directory and copy the file /software/Aglets1.1b3/examples/hello/HelloAglet.java into it.
2. cd into your aglets directory (the parent directory of the examples directory) and compile the HelloAglet, javac examples/hello/HelloAglet.java.
3. Create two Tahiti servers listening on different ports. You need to first open xterm or command windows for each server. For example, on jupiter (Solaris) type **xterm &**. With my setup I could then type, **agletsd -f myaglets.props** and **agletsd -f myaglets.props -port 8888**. The first command sets up Tahiti to listen on the port specified in the property file. (You must be running X Windows on solaris (or Linux), or Win 95/98/NT. The HelloApplet will not work with the command line version of Tahiti.)
4. In one of theTahitis, click the Create button. From the list that appears, select examples.hello.HelloAglet and click the Create button. A line in the Tahiti window notifies you that the Aglet is present.
5. A pop up window also appears. In its Address field type in the address of the other Tahiti server. For example, atp://jupiter.scs.ryerson.ca:9000, or atp://localhost:9000. You can add this address to the address list if you wish.
6. Finally, click the go button. The aglet should dispatch to the other Tahiti window where it prints out "Hello, world". It stays there for a few seconds and returns to its origin Tahiti where it prints out "I'm

back", and then disposes of itself.

7. You can, of course, also send your HelloAglet to a Tahiti server on another computer on the SCS network. For example, proton.scs.ryerson.ca which listens on the default port 4434. This is a NT machine.

Example 3. CirculateAglet.

This aglet travels from Tahiti server to Tahiti server collecting information at each stop. It finally returns to its origin server and prints out what it has learned to stdout (an xterm or command window). You will have to modify the supplied code to have the addresses of available servers.

1. In your examples directory, create a subdirectory called itinerary. Then copy all the files from /software/Aglets1.1b3/examples/itinerary into it. These files belong to the package examples.itinerary.
2. Using your favourite editor edit the file CirculateAglet.java. Locate the three "addPlan" methods. Change only the first two of these. If you have the same Tahiti servers running as in example 1, change the address in one of these addPlans, to say, atp://localhost:9000, and the other to atp://proton.scs.ryerson.ca. You might also replace the getProxy() with a second getLocalInfo() which produces more interesting information.
3. Because the code is in a package, examples.itinerary, cd back to the parent directory of examples (directory aglets in my case) and compile with the command **javac examples/itinerary/*.java**. (This also compiles another example in the package.)
4. In the origin Tahiti window (the one that is not listening on port 9000 in our examples), create the itinerary aglet using the Create button and selecting **examples.itinerary.CirculateAglet.**
5. The Tahiti window shows that the CirculateAglet is loaded. Select it and then click the Dialog button. A window pops up showing the places the Aglet will visit.
6. Click the Start! button. The Aglet appears briefly in the other Tahiti window on your machine and then takes off for proton.scs.ryerson.ca. It returns to its original window. Iif you look in the corresponding xterm window you should see some properties of the machines it visited. You can dispose of the returned aglet by selecting it in the Tahiti window and clicking the dispose button.

Note on Win 95/98/NT etc.

You run these examples in the same way as on UNIX. One thing to watch out for. You need to double the backslashes in path name strings in Java on these systems to get the path backslash because a single backslash is an escape character in Java.

SCS NT labs

Aglets are not set up in these labs. Use Xwin32 and run on jupiter.

Linux

Setting up should be the same as on NT. Just unzip the distribution and run ant. I did find one problem. The ant file I had was a DOS type file with CR as well as LF as line terminators. I had to remove the CR's before ant would work.

The Aglet Model

Basic Elements

- **Aglet**. a mobile Java object running in its own thread. Reactive to its environment. Has a degree of autonomy. *com.ibm.aglet.Aglet*
- **Proxy**. Represents the aglet. Protects the aglets's public method. Provides location transparency for the aglet. The aglet's proxy does not move.

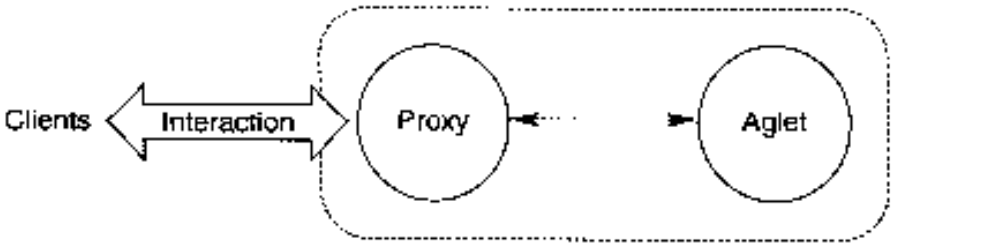


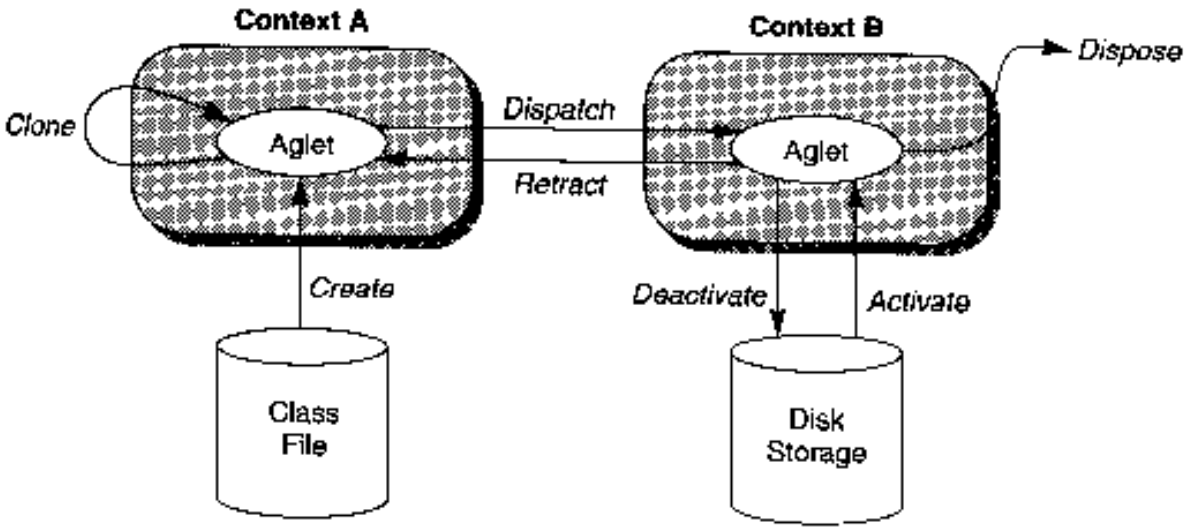
FIGURE 3-1 Relationship between Aglet and Proxy

com.ibm.aglet.AgletProxy

- **Context**. The aglet's place or environment. Provides services for the aglet and protects the host from malicious aglets. Contexts are named, and thus can be located by their host, port and name. *com.ibm.aglet.AgletContext*
- **Identity**. A unique, network wide ID for an aglet.

Fundamental Operations

- **Creation**. The aglet is loaded into a context. It is initialized and immediately begins execution.
- **Cloning**. A second way of creating an aglet in a context. An aglet already there is copied. The clone has its own identifier and starts running at once. Threads are not cloned so the cloned aglet stops executing.
- **Dispatching**. Moves the aglet to a new context. Its thread does not go with it. Upon arrival it starts running in a new thread from its starting point. (The Java instruction pointer position cannot be remembered.)
- **Retraction**. Recalls an aglet from its present context to the context and inserts it into the context from which the retraction call was executed.
- **Activation/Deactivation.** An aglet can be stopped and temporarily transferred (serialized) to disk. After a certain time the aglet can be activated again into the same context.
- **Disposal.** Gets rid of the aglet and allows the Java garbage collector to remove its remains.



GURE 3-3 Aglet Life-Cycle Model

The Aglet Event Model

Aglets use the delegation event model introduced in the JDK1.1. The Aglet model adds three event listeners.

- **Clone Listener**. Listens for cloning events.
- **Mobility Listener**. Listens for events generated by aglet movements. This is the most important listener. You use it to take action when an aglet is about to be dispatched, when it arrives, and when it is retracted. *com.ibm.aglet.event.MobilityEvent*
- **Persistence Listener**. Listens for activation/deactivation events.

The Aglet Communication Model

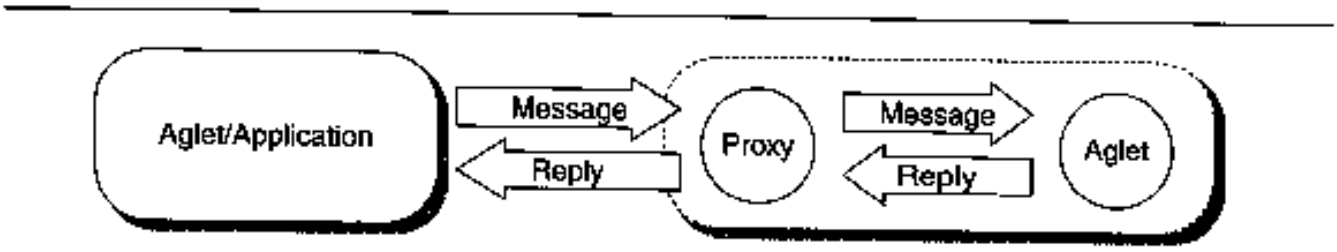


FIGURE 3-5 Aglet-to-Aglet Messaging

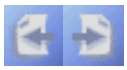
Note that messages go via the aglet's proxy.

Aglets have three types of message.

- **Message**. A message is an object exchanged between aglets. The basic message mechanism is synchronous. The sender waits for a reply. There is also a simple asynchronous mechanism, the one way message, when no reply is needed.
- **Future Reply**. It is also possible to have asynchronous messages when a reply is needed. The sender carries on while waiting for a reply in the future.
- **Reply Set**. A reply set can contain multiple future replies.

See also.

CERN CSC99 Agents/2



Aglet Mobility

The Aglet system emphasizes mobility. Although it supports inter-aglet messaging, as it must, the message support is much less elaborate than some other systems such as JADE.

Moving Aglets Around

The Aglet API provides two main methods

- retractAglet()
- dispatch()

The Tahiti server also implements dispatching and retracting so the easiest way to move an Aglet is to create it in Tahiti and then click the approriate buttons!

However, as mentioned in the section on remote messaging, the more usual arrangement is to have a parent, stationary, Aglet which creates and dispatches child Aglets to remote locations. The parent can also retract the children from these locations.

Dispatching

There are a number of versions of the dispatch() method

In the AgletProxy Interface

There are two methods. The most commonly used one is,

AgletProxy dispatch(java.net.URL url)

This method is usually called in one Aglet to dispatch another. The first aglet keeps track of the proxy of the dispatched aglet.

In the Aglet class

Again there are two versions. The most commonly used is,

void dispatch(java.net.URL url)

This method allows aglets to dispatch themselves to a new location.

Retracting

There is only one method to do this.

In the AgletContext interface

AgletProxy retractAglet(java.net.URL url, AgletID id)

Arrival and Departure

Two interesting events in the life of an Aglet are arriving and departing. Naturally, the Aglet API takes advantage of the Java Event model to handle these situations.

To handle these events a new Java event, MobilityEvent has been added. There is a corresponding MobilityListener and a MobilityAdapter.

Three methods must be implemented by a MobilityListener, onArrival(), onReverting(), and onDispatching(). For notes on these methods, of which onArrival() is especially important, click on the following link.

Notes on the Aglet Event Model

A basic mobility example.

[top] [previous] [next] Questions?

Aglet Event Model

The JDK Event Model

The aglet API uses the Java delegation event model introduced in the JDK 1.1. In this model, certain objects generate events when certain actions are taken on them. The simplest example is the AWT Button object which generates an ActionEvent object when a button object is clicked with a mouse. Java has many objects which generate events. You can also have your own objects generate events if you so desire.

Only objects registered with the event generating object respond to events. Other objects pay no attention. They are not interested. Registered objects which are interested are said to listen for events.

Registration implies an obligation on the part of the listener object. The registered object must be prepared to handle the events generated by the object with which it registered. Just what an object has to handle when it registers with an event generating object depends on the type of object generating the events.

In the case of an ActionEvent, you need only do one thing. You must perform actions by implementing the method `public void actionPerformed(ActionEvent)` in the listening object's class definition.

On the other hand, to respond to a WindowEvent generated by an object of the Window class (or its subclasses such as Frame), you must implement 7 different response methods in the listener.

Aglet Generated Events

Since aglets are written in Java, they may be involved with any of the Java events. They add three events of their own:

- MobilityEvent
- CloneEvent
- PersistencyEvent

Of these, the MobilityEvent is the most used. It is the only one covered in cps720.

Dealing with Mobility Events

Mobility events are generated by aglet motion. There are three situations covered.

- dispatch. The aglet is dispatched to another location
- arrival. The aglet arrives at another location
- retraction. The aglet is called back to another location.

To each of these there corresponds a method which must be implemented by any object listening for

mobility events. These methods are:

- **`void onDispatching(MobilityEvent).`** Called when the aglet is being dispatched
- **`void onArrival(MobilityEvent)`**. Called just after the aglet arrives at its destination, and before its run() method is executed
- **`void onReverting(MobilityEvent)`**. Called on the remote host just before the aglet moves back to where is being retracted to.

Of these, the onArrival() method is used most often.

These methods belong to the MobilityListener interface. Because the MobilityListener interface is an interface not a class, all three methods must be implemented by any class which implements the interface. (You can get around this stipulation by using the MobilityAdapter class. This is discussed later.)

MobilityEvent Methods.

The MobilityEvent class has two useful methods.

- **`AgletProxy getAgletProxy()`**. Returns the proxy of the aglet which generated the mobility event.
- **`URL getLocation()`**. Gets the location (the server) where the aglet generated the event.

A Simple Aglet Mobility Example

This example has two Aglets. One is the "master" aglet. It is loaded into the server with the Create button. This master Aglet creates a child Aglet and sents it to another server. On arrival at the second server the mobile Aglet prints a simple message on the server console, and then dispatches itself back to the original sever where it prints the same message on its home console.

In these examples, red = method called by the Aglet system, blue = important Aglet methods, and green - important Aglet classes.

The Master Aglet

BasicMasterAglet.java source

```
package cps720.mobile;

import com.ibm.aglet.*;

import java.net.*;

public class BasicMasterAglet extends Aglet {

    public void onCreation(Object init) {

        try {

            AgletContext ac = getAgletContext();

            URL homeBase = ac.getHostingURL();

            AgletProxy mobileAgletProxy = ac.createAglet(null,

                "cps720.mobile.BasicMobileAglet",

                homeBase);

            mobileAgletProxy.dispatch(new
            URL("atp://jupiter.scs.ryerson.ca"));

        } catch (Exception ex) {

            ex.printStackTrace();
```

```
            }
        }
    }
```

Notes

- This example introduces three very important classes: Aglet, AgletProxy, and AgletContext.
- The onCreation() method is called by either the Aglet server (Tahiti) as here, or in response to a createAglet() call, as in the case of the mobile agent.
- The Object argument of onCreation() allows the newly created Aglet to receive initial state information. The Object referenced by 'homeBase', the third argument to the createAglet() method, appears as the Object argument in the onCreation() method of the created Aglet. (See the BasicMobileAglet below.)
- The getHostingURL() method belongs to the AgletContext class. There is also a method called getAddress() which returns a String.
- If you look in the documentation you will see that many Aglet methods throw multiple exceptions. You can cover all thses with the plain Exception class. On the other hand you may loose precision doing this if you are debugging.

The Mobile Agent

BasicMobileAglet.java source

```
package cps720.mobile;

import com.ibm.aglet.*;

import com.ibm.aglet.event.*;

import java.net.*;

public class BasicMobileAglet extends Aglet {

    private URL homeBase = null;

    public void onCreation(Object initInfo) {

        homeBase = (URL) initInfo;

        addMobilityListener(new MobilityAdapter() {
```

```
                public void onArrival(MobilityEvent me) {
                    setText("Phew, arrived safely");
                }
            });
        }
        public void run() {
            try {
                Thread.sleep(20000);
            } catch (InterruptedException ie) {
            }
            URL whereAmI = getAgletContext().getHostingURL();
            if(!whereAmI.equals(homeBase)) {
                try {
                    dispatch(homeBase);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
}
```

Notes

- Notice how the home base address is passed to the mobile child agent from the master agent when it is created.
- Using an anonymous inner adapter class is an important idiom when programming Aglets.

Exercise

This program writes the same message both when the mobile Aglet reaches the remote server and when it retruns home. Suppose you wanted to write a different message when arriving home? How would you do it? Also, how would you have the mobile Aglet dispose of itself after writing this return message?

Usage

This program works fine on simple lan (one subnet) or if the Aglet server can resolve the fully qualified name of the servers (for example, jupiter.scs.ryerson.ca, not just jupiter). If this resolution fails, the mobile Aglet cannot dispatch itself home. However, it can be retracted in such situation. If you have a home PC you might try experimenting with this.

```
package cps720.mobile;

import com.ibm.aglet.*;
import java.net.*;

/**
 * A parent stationary Aglet which launches a child mobile agent to
 * another server.
 * To run, change the target URL to something suitable for your
 * system and recompile.
 * See also BasicMobileAglet.java
 * DG. Sept. 2001
 */
public class BasicMasterAglet extends Aglet {

    String targetServer = "atp://IBM:9000";

    public void onCreation(Object init) {
        try {
            AgletContext ac = getAgletContext();
            URL homeBase = ac.getHostingURL();

            AgletProxy  mobileAgletProxy = ac.createAglet(null,

"cps720.mobile.BasicMobileAglet",
                                                          homeBase);
            mobileAgletProxy.dispatch(new URL(targetServer));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

```
package cps720.mobile;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;


/**
 * A simple movile agent launched by BasicMasterAglet.
 * This agent, once launched goes to the target server, writes a message
 * and then dispatches itself back to the sender if possible (that is, if
 * it has the fully qualified address of the sender host, including the
 * domain.
 * see also BasicMasterAglet.java
 * DG. Sept. 2001
 */
public class BasicMobileAglet extends Aglet {

    private URL homeBase = null;

    public void onCreation(Object initInfo) {

        homeBase = (URL) initInfo;

        addMobilityListener(new MobilityAdapter() {
            public void onArrival(MobilityEvent me) {
                setText("Phew, arrived safely");
            }
        });
    }

    public void run() {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException ie) {
        }
        URL whereAmI = getAgletContext().getHostingURL();
        if(!whereAmI.equals(homeBase)) {
            try {
                dispatch(homeBase);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Remote Messaging

Aglets can exchange messages across the network. You can have fixed Aglets which interact this way. Of course, then you no longer have a *mobile* agent system. Only data moves (the messages) not the code (the agent).

Nevertheless, remote messaging is an integral part of the Aglet mobile agent system. A frequently used pattern is to have an agent move to a remote host, do some intensive calculations, send the result back as a message, and dispose of itself.

Some difficulties with remote messaging

There are two main difficulties

- Getting appropriate proxies and ID's
- Network latency

Network Latency

In other words, the Net can be slow. This may or may not matter too much but it could leave an Aglet hanging, waiting for a reply to a message. The Aglet API provides the Future Message to do messaging asynchronously and avoid hanging.

Getting Proxies and ID's

As with local messages, communicating aglets must know one another's proxies and to get these, the Aglets may also have to kwow their ID's. Obtaining this information when all the Aglets are in one context (the local situation) is straigthforward. Obtaining the information across a network is not so easy.

If you have the requisite proxy and ID information, then sending messages to remote Aglets across the network is done the same way as locally.

Contacting Remote Aglets

Actually, remote agent communication is not so easy. Suppose,for example, that two people set up tahitis on two separate machines (both on the default port, 4434) and created an Aglet on each server. How could these two Aglets communicate? Each Aglet does not know the proxy or ID or the other.

One side would have to create some kind of "messanger" Aglet to go to the other server, ask the Aglet there for its proxy and ID and send these back to the original Aglet at its home base (or go back itself with this information). The messanger presumably be carrying the proxy and ID of the Aglet which dispatched it. With both sides having the requisite information, assuming they had some kind of common language, they could communicate with messages.

The pattern of one stationary Aglet sending another to a remote server is a common one. The stationary Aglet is sometimes called the master, the mobile Aglet the slave, or parent and child.

The parent keeps track of the proxies and ID's of its children. Similarly, the children keep a record of the parent's proxy and ID. Two way remote messaging becomes possible.

If the mobile, child Aglet can decide on its own to change its location, the problem of tracking comes up. The parent may lose contact with the child. This can be a difficult problem.

The examples illustrating remote messaging also illustrate the various mobility mechanisms provided by the Aglet API. Tor understand the examples, you need to know something about these mobillity mechanisms. These are reviewed next.

[top] [previous] [next] Questions?

Aglet Local Messaging

A common pattern in Aglet programming is the meeting pattern. An Aglet arrives in a context where other Aglets exist and exchanges local messages with them. Since messages are local, the network is not cluttered up with them. This advantage can be important if there is a lot of interaction between Aglets.

Basic local messaging example

We continue with the SayIt and HearIt Aglets. This time they use messages to communicate. (SayItAglet must be loaded into Tahiti first.)

Here is the new version of SayItAglet. SayItAglet3.java

```
package aglets.mystuff.testtalk;
import com.ibm.aglet.*;

public class SayItAglet3 extends Aglet {
        private String [] msg = new String[2];

        public void onCreation(Object init) {
                msg[0] = new String("David Grimshaw");
                msg[1] = new String("Hello");
        }

        public String [] getmsg() {
                return msg;
        }

        public boolean handleMessage (Message msg) {
                if(msg.sameKind("Who are you?") ){
                        msg.sendReply(getmsg());
                        System.out.println("Message: Who are you? received.");
                        return true;
                }
                return false;
        }
```

```
}
```

What's new here?

This example features an object (reference) of the Message class, and two of that classes's important methods, sameKind() and sendReply(). It also shows the very important method of the aglet class, handleMessage().

Aglet class

- **boolean handleMessage(Message)**
  This is a key method for receiving messages. Note that it returns a boolean. A return of true indicates that the message has been dealt with. A return of false indicates that this aglet is ignoring the message.In other words, the aglet does not understand the message.

Message class

- **boolean sameKind(String)**
  This method allows the receiving aglet to distinguish among messages the aglet wants to understand and respond to. Its argument is a message key to be compared with keys of incoming messages.
- **void sendReply (Object)**
  The easiest way to reply to a message is to use the Message object itself to carry a reply to the sender of the message.

A new version of the HearItAglet. HearItAglet3.java

```
package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import java.util.*;

public class HearItAglet3 extends Aglet {

    public void run() {
        Aglet anAglet = null;
        AgletProxy ap = null;
        AgletContext ac = getAgletContext();
        for(Enumeration aps = ac.getAgletProxies(); aps.hasMoreElements(); ) {
            ap = (AgletProxy) aps.nextElement();
            try {
                anAglet = ap.getAglet();
```

```
                    System.out.println(anAglet.getAgletID());
                } catch (InvalidAgletException iae) {
                    System.out.println("Invalid aglet");
                }
                if(anAglet != null) {
                    String agClassName = anAglet.getClass().getName();
                    if(agClassName.equals("aglets.mystuff.testtalk.SayItAglet3")) {
                        Message msg = new Message("Who are you?");
                        String [] itSaid = new String[2];
                        try {
                            itSaid = (String []) ap.sendMessage(msg);
                        }
                        catch (Exception e) {
                            itSaid[0] = "unknown"; itSaid[1] = "unknown";
                            System.err.println(e.toString());
                            return;
                        }
                        setText(itSaid[0] + " " + itSaid[1]);
                    } else {
                        System.out.println("Another aglet present");
                    }
                } else {
                    setText("No one understand me.");
                }
            }
        }
    }
```

What's new here?

In the Message class

- **Message(String)**
  The simplest of the many overloaded Message constructors. There is no message value defined. The String argument is used to identify the message for the sameKind() method in the receiver aglet. Can be used just as a notifier (as above), or with setArg(String, Object) to send a Hashtable message (see below).

In the AgletProxy class

- **Object sendMessage(Message)**
  Another important method. The object returned is a possible reply message from the receiver. There is also void sendOnewayMessage(Message) if you are not interested in a reply, and the more complex, FutureReply sendFutureMessage(Message).

The Message Class

Aglet messages are Java objects belonging to the message class. Objects of this class carry a lot of information. A message object is either a key-value pair, or it can be a hash table.

Key-value pair type of message

The message key is always a string, such as, "Who are you?" in the example. The value can be any primitive type (int, float, etc), or an object of type Object, or any subclass of Object. In other words, any Java object will do. The value (argument) can be retrieved using the Message class's **`getArg()`** method.

In the case of messages among objects at the same place, any Java object will indeed do. If the messages are sent by aglets in different places, however, there is a limitation upon what objects can be sent in messages. A message object sent to a remote host must be serializable. Not only that, any objects referenced in the non-transient fields of the object must also be serializable. And any objects referenced by these objects must also be serializable. And so on.

Verifying that these chains of references refer only to serializable things can be tricky. The simplest way to handle this situation is to use Strings, or Vectors or Arrays of Strings as the values of messages, if at all possible.

Creating simple messages

Message msg = new Message("Dave's Message", "Hello");	Message msg = new Meassge("answer", 42)

Hash table messages

You can create more elaborate messages using this feature of the aglet API. First you create a Message object with a message key, then you put data in the message hash table using setArg(key, value). The receiver can use the keys to retrieve values with getArg(key).

A simple example.

```
Message msg = new Message("constants");
msg.setArg("pi", 3.1416)
msg.setArg("myname", "Dave");
```

The receiver would decode the message something like this:

```
boolean handleMessage(Message msg) {
```

```
            if(msg.sameKind("constants") {
                float shortPi = ((Float) msg.getArg("pi")).floatValue();
                String name = (String) msg.getArg("myname");
                // do something
                return true;
            }
            return false;
        }
```

The Message class has many other features, some of which will be discussed later.

Message Class

[top] [previous] [next] Questions?

Inter-Aglet Communication

In multi-agent systems, inter-agent communication is an important topic. Aglets are no exception. The Aglet API provides a number of communication mechanisms. However, Aglets provide only limited support for message content. Content is just objects, usually strings or vectors of strings.

There are several ways for Aglets to communicate with other Aglets.

Direct Communication Among Aglets in the Same Place

If Agelts are in the same place (server) they can communicate without messages. There are two ways to do this. One Aglet can call the public methods of another via its proxy. The other way is for one aglet to leave a message in the environment to be picked up by another Aglet when it arrives. The first of these methods is not recommended at all as it violates the integrity of Aglets. It's rather like being able to read some else's mind instead of listening to her/him talk!

Examples of direct communication

Communication via Messages

This is the preferred method of inter-aglet communication. The message mechanism preserves the integrity of Aglets and does not clutter up the environment.

Messages can be sent among Aglets in the same place, that is, locally, or to Aglets at remote contexts. The Aglet API is designed to minimize the differences between programming local and remote messaging.

There are several different kinds of Aglet messages,

- two way messages
- one way messages
- future (asynchronous) messages

Messages fall into two categories, synchronous, and asynchronous. With the former, the sender waits for a reply. With the latter, the sender goes on to something else while waiting for the reply, and then deals with the reply when it comes in.

Local Messages

Example and Discussion

Remote Messages

Example and Discussion

Asynchronous Messages

Example and Discussion

Mobility and Messaging Examples

An Example from the textbook and another SayIt/HearIt example

A Basic Remote Messaging Example

An exampe of an Aglet and its itinerary

The need for semantics

The simple string comparing methods used by the aglet API are bound to limit the intelligence of agents. For intelligent agents, communication mechanisms that allow semantic content will ultimately be needed. In other words, higher level communication languages are needed.

Two languages that allow for semantic content in messages are,

- SL (and KQML). SL, semantic language, is FIPA's simplified and standardized version of KQML, Knowledge Query and Manipulation Language, originally developed in the USA under the auspices of the Department of Defence. These languages are based on Speech Act theory.
- XML. Extensible Meta Language.
-
- In addition, for meaningful communication, ontologies must be developed.

These ideas are discussed later in the course.

Direct Communication Among Aglets in the Same Place

Communication by public method invocation

If Aglets are in the same location, they can 'communicate' by calling one anothers public methods. Although this approach is sometimes convenient, it is not recommended because it violates the encapsulation principle. The agents (Aglets) are no longer independent and autonomous.

Calling public methods examples. (both examples use this method)

Communication via the Environment

Aglets can also commuicate by modification of their environment. This method also occurs in nature. Ants leave scent trails. Dogs mark fire hydrants. In the case of Aglets, one Aglet can leave information in an environment to be picked up later by another Aglet arriving in the same environment.

Communication using the environment example. (Example 1 only)

Inter-Aglet Communication without Messaging

These examples illustrate two ways of inter-aglet communication for Aglets in the same place (location, server).

- Calling one another's public methods
  In this case the Aglets must be in the same location at the same time. This method is not recommended for many reasons,
- Picking up information left in the environment by other Aglets..
  In this case, the Aglets do not have to be in the same location at the same time. Usually they are not. This method can be useful is used with restraint. The danger is that the environment will be filled up with leftover garbage.

This series of examples involves two stationary Aglets, SayitAglet and HearItAglet. The former wants to "say something" to the latter.

In these examples one aglet arranges to invoke a public method of another aglet.

You cannot call aglet public methods directly. You must access them via the aglet's AgletProxy class. Also, to access its public methods, you must also know the aglet's ID.

1. Using the AgletContext's property facility to communicate an aglet's ID.

In this example, one Aglet leaves its ID in the environment and waits. Another Aglet which arrives in the same context can retrieve this ID from the context (i.e., from the environment) and use it to get the waiting Aglet's proxy. It then calls the waiting Aglet's getMsg() public method via the proxy in order to find out what was said.

(Of course, you could, more simply, arrange to put the message itself into the context as a property for the second Aglet to get. That would avoid the need to know the name of the Aglet's public method, getMsg().)

SayItAglet.java

```
package aglets.mystuff.testtalk;
import com.ibm.aglet.*;

public class SayItAglet extends Aglet {
    private String [] msg = new String[2];

    public void onCreation(Object init) {
        msg[0] = new String("David Grimshaw");
        msg[1] = new String("says Hello");
        AgletContext ac = getAgletContext();
        ac.setProperty("SayItID", getAgletID());
    }
    public String [] getmsg() {
        return msg;
    }
}
```

Aglet notes:

- The message is an array of Strings, for no particular reason, other than it is often useful to carry information with your aglet in an String array or vector.
- The onCreation() method is analogous to the init() method of an applet. It is called once when the aglet is constructed. You do not normally call the actual aglet constructor. Use create() to initialize your aglet if necessary. Your version overrides the base version of create() in the Aglet class.

Java notes:

- It is a good idea to put your aglets in a package since they often contain multiple files. Packages are a bit tricky. See the cps840 notes on packages, or the Java Tutorial.

Important aglet methods

In the Aglet class

```
AgletContext getAgletContext()
```

The aglet gets a reference to its current environment.

```
AgletID getAgletID()
```

The aglet knows its own "name" (a bignm).

In the AgletContext class

```
void setProperty(String key, Object property)
```

Sets a property in the environment. Since the property is an object of the Object class it can be anything except a primitive. (You can wrap the primitives.)

```
Object getProperty(String key)
```

You use the String index to retrieve the property. Since the result is an Object object, you have to cast the return value into the actual property class type.

Aglets often leave property values in a context as a kind of "fingerprint", to be picked up by another aglet later. Our SayItAglet leaves its ID as a fingerprint for HearItAglet to pick up.

Now consider this aglet, which arrives later in the context

HearItAglet.java

```
package aglets.mystuff.testtalk;
import com.ibm.aglet.*;

public class HearItAglet extends Aglet {
    public void run() {
        AgletContext ac = getAgletContext();
        AgletID aid = (AgletID) ac.getProperty("SayItID");
        AgletProxy ap = ac.getAgletProxy(aid);
```

```
            Aglet sayItAglet;
            try {
                sayItAglet = ap.getAglet();
            } catch (InvalidAgletException iae) {
                sayItAglet = null;
            }

            if(sayItAglet != null) {
                String [] itSaid = ((SayItAglet)sayItAglet).getmsg();
                setText(itSaid[0] + " " + itSaid[1]);
            } else {
                setText("Oops .. no Say It aglet here.");
            }

        }

    }
```

Aglet Notes

- **run().** Aglets run in their own threads. Therefore, like any other thread they are supplied with their own run() method which you can override if you wish to. The run() method is only executed after the onCreation() method returns. We just used the Aglet class version of run() in SayItAglet and overrode onCreation() for our own purposes. In HearItAglet we do the reverse. It is important to note that when the aglet is dispatched to a remote context, the run() method restarts from the beginning (after the onArrival() method is executed).
- You can use the Aglet class's `setText()` method to put simple one line messages on the Tahiti console. For debugging simply use System.out.println() to direct output to the console.
- Note the **sequence of actions**. First the HearItaglet gets a reference to its context. Then it uses the getProperty() method to get the ID left by the SayITAglet in the context. Now that it has the SayItAglet's ID, the HearITAglet can get a reference SayItAglet's proxy. Using this reference we can get (a reference to) the aglet itself and use it to call the public getmsg() method of the SayItAglet. The message appears in the Tahiti window.

Java Notes

- **Exceptions**. Java can throw many exceptions, especially the network code. Some of these, called "checked" exceptions, must be caught (i.e.., dealt with), usually using the try/catch mechanism.
- **try/catch** scope. The scope of things declared in a try block is the try block alone. So, for example, if you were to declare sayItAglet inside the try, you would get an undeclared variable error at the reference to sayItAglet after the catch.
- try/catch and initialization. If you left the line sayItAglet = null out of the catch block, the code does not compile because there is a possibility that sayItAglet would not be initialized when it is referenced in the line `if(sayItAglet != null) {.` (The execution could go through the catch if getAglet() failed.) Of course, you could just have `Aglet sayItAglet = null;` instead of `Aglet sayItAglet;` in the line above the try.

Important Aglet Methods

In the AgletContext class

```
AgletProxy getAgletProxy(AgletID id)
```

Very useful. But you need to know the AgletID to use it.

In the AgletProxy class

```
Aglet getAglet(AgletProxy ap)
```

Once you have the proxy, you can get a reference to the aglet itself and access its public members.

---

2. Using the getAgletProxies() method to get an aglet's ID

In this approach, the context is left alone. An enumeration of all the proxies in the context is made. Then you have to find the proxy you want from among them. This example, like the previous one, uses the proxy to call the SayItAglet's public getMsg() method.

SayItAglet2.java

This is even simpler than SayItAglet.java. It is created and then just sits there. Even though its thread finishes after creation, it is not garbage collected.

```
package aglets.mystuff.testtalk;

import com.ibm.aglet.*;

public class SayItAglet2 extends Aglet {
    private String [] msg = new String[2];

    public void onCreation(Object init) {
        msg[0] = new String("David Grimshaw");
        msg[1] = new String("Hello");
    }

    public String [] getmsg() {
        return msg;
    }
}
```

HearItAglet2.java

This is more complicated than HearItAglet.java because of the need to determine which of the proxies one wants to deal with.

```
package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import java.util.*;
```

```
public class HearItAglet2 extends Aglet {
    public void run() {
        Aglet anAglet = null;
        AgletContext ac = getAgletContext();
        for(Enumeration aps = ac.getAgletProxies(); aps.hasMoreElements(); )
        {
            try {
                anAglet =
                ((AgletProxy)aps.nextElement()).getAglet();
                System.out.println(anAglet.getAgletID());
            } catch (InvalidAgletException iae) {
                System.out.println("Invalid aglet");
            }
            if(anAglet != null) {
                String agClassName = anAglet.getClass().getName();
                if(agClassName.equals("aglets.mystuff.testtalk.SayItAglet2"))
                {
                    String [] itSaid = ((SayItAglet2)anAglet).getmsg();
                    setText(itSaid[0] + " " + itSaid[1]);
                } else {
                    System.out.println("Another aglet");
                }
            } else {
                setText("Oops .. no Say It aglet here.");
            }
        }
    }

}
```

Java Notes

- Enumeration interface. This interface is in the package java.util. An enumeration is a list which can be traversed once, usually using a for loop as shown above. The two most important methods are
  - `boolean hasMoreElements()`
  - `Object nextElement()`
  - Note that nextElement() returns a generic object which must be cast to the appropriate type before use.
- The Class class. Every Java class has a Class object associated with it which you can use to obtain information about the class. The HearItAglet2 aglet shows a simple use of this Class. You can do all kinds of fun things with the class Class when combined with the java.lang.reflect package. The method getClass() returns a Class object associated with the Aglet object. The method getName() of the Class class gets the name of the HearAglet2 class as a string. getName() returns the "fully qualified" name.

Important aglet methods

In the AgletContext class

```
Enumeration getAgletProxies()
```

If you have the context, you can get references to all the proxies of the aglets "living" there. Of course, you need to know how to use the Java Enumeration interface.

Moving the aglets around.

The four aglets discussed above cannot move by themselves but you can move them using Tahitt's dispatch and retract commands. You will find after playing with this for a while that the second pair is more robust than the first.

Using context properties to leave "fingerprints" also pollutes the environment. The property remains until it is deliberately set to null, or the server is rebooted.

For example, if you send SayItAglet to another server and then load HearItAglet, HearItAglet will pick up the ID of the departed aglet but the aglet is no longer there. You get a null pointer exception,

On the other hand, if you send SayItAglet2 off to another server, then crate HearItAglet2 on the first server, there is no error. You can then dispatch HearItAglet2 to the second server, catching up to SayItAglet2, and it correctly calls SayItAglet2's getmsg() method.

Note that HearItAglet2 did not "see" the departed SayItAglet2 when it is created and run. To communicate with thse methods, the two aglets must be in the same context.

```
/**
 * sayItAglet3.java
 * Inter aglet communication.
 * This aglet is perpared to receive a "Who are you?" message and reply to
 * it.
 */

package aglets.mystuff.testtalk;

import com.ibm.aglet.*;

public class SayItAglet3 extends Aglet {
        private String [] msg  = new String[2];

        public void onCreation(Object init) {
                msg[0] = new String("David Grimshaw");
                msg[1] =  new String("Hello");
        }

        public String [] getmsg() {
                return msg;
        }

        public boolean handleMessage (Message msg) {
                if(msg.sameKind("Who are you?") ){
                        msg.sendReply(getmsg());
                        System.out.println("Message: Who are you? received.");
                        return true;
                }
                return false;
        }
}
```

```
/**
 * HearItAglet3.java
 * This sends a message to SayItAGlet3
 * You don't need to check which aglets receive the message, since those
 * which do not understand the message do not reply.
 * On the other hand, this is a waste of energy so it is probably a good idea
 * to be selective anyway. (Below the selection method used in HearItAglet2 has been
 * commented out.
 * DG Sept. 99
 */

package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import java.util.*;

public class HearItAglet3 extends Aglet {

        public void run() {
                Aglet anAglet = null;
                AgletProxy ap = null;
                AgletContext ac = getAgletContext();
                for(Enumeration aps = ac.getAgletProxies(); aps.hasMoreElements(); ) {
                        ap = (AgletProxy) aps.nextElement();
                        try {
                                anAglet = ap.getAglet();
                                System.out.println(anAglet.getAgletID());
                        } catch (InvalidAgletException iae) {
                                System.out.println("Invalid aglet");    // otherwise
compiler complains of undefined variable
                        }
                        if(anAglet != null) {
                                String agClassName = anAglet.getClass().getName();
                                        Message msg = new Message("Who are you?");
                                        String [] itSaid = new String[2];
                                        try {
                                                itSaid = (String []) ap.sendMessage(msg);
                                        }
                                        catch (Exception e) {
                                                itSaid[0] = "unknown"; itSaid[1] =
"unknown";
                                                System.err.println("Message or answer
failed.");
                                        }
                                        setText(itSaid[0] + " " + itSaid[1]);
                        } else {
                                setText("Oops .. no Say It aglet here.");
                        }
                }
        }
}
```

Remote Aglets

When Aglets are dispatched to remote sites they must be tracked. The mobile Aglets must also keep track of where their home is. Furthermore, they may need to know enough location information to send messages anong one another.

Different mobile agent API's do this in different ways. The Aglet API has evolved over several years during which changes have occurred in how Aglets are kept track of. It is not such a good idea to make Aglets too easy to track since a hostile aglet might take control of your Aglet. Convenience vs. security is an old story on the Internet!

Several changes have been introduced in the Aglet API between versions 1.0 and 1.1. Both changes have been made for reasons of security.

Two changes introduced in the API 1.1

- The AgletContext method, **`AgletProxy getAgletProxy(URL, AgletID),`** is *deprecated*. This term means that you can still use it but the method may be removed in the next release, thereby breaking any code you have written using it. However, as of October 2000 this method was still available. Note that there is a second version of getAgletProxy, AgletProxy getAgletProxy(AgletID id) which is not deprecated and can be used in a local context.
- The important AgletProxy method, **`AgletProxy dispatch(URL)`** still returns the proxy of the aglet when it reaches the remote place, but the proxy is invalid in API 1.1 and cannot be used. (It could be used to interact with the remote aglet in version API 1.0. because it was valid.)

Significance

Both these methods were used to obtain a reference to the remote aglet's proxy. As you know, to do much with an aglet, you need to have a reference to its proxy. The changes listed above eliminate two easy ways of getting a reference to a remote aglet's proxy.

Version 1.1 of the aglet API has substituted other ways of contacting remote aglets. Section 7.3 of Oshima's and Lange's book describes a basic aglet finder (p.126-130). A more sophisticated finder is included in the 1.1 package. See the documentation.

It is still possible to do useful messaging with remote aglets without using the finder. See the second example below.

The new version 1.1 response of the dispatch() method is illustrated by the following example from the book.

Aglet Book Example

This example, taken from Chapter 7 of Lange and Oshima (p. 121) illustrates the second difference mentioned above. There are two classes:

ProxyDispatchExample.java

```
/*
 * @(#)ProxyDispatchExample.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima,  1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
```

```
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */
package aglets.agletbook.chapter7;
import com.ibm.aglet.*;
import java.net.URL;

public class ProxyDispatchExample extends Aglet {
   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         AgletProxy proxy = getAgletContext().createAglet(getCodeBase(,
,"aglets.agletbook.chapter7.ProxyDispatchChild",
,
                                                          null);
         print("Finished creating the child.");
         pause();
         print("Proxy Valid:        \'" + proxy.isValid() + "\'");
         print("Proxy Remote:        \'" + proxy.isRemote() + "\'");
         print("Dispatching the child...");
         String host = getAgletContext().getHostingURL().toString();
         URL destination = new URL("atp://localhost:9000");
         print("Destination is \'" + destination.toString() + "\'");
         AgletProxy remoteProxy = proxy.dispatch(destination);
         print("Finished dispatching the child.");
         pause();
         print("Proxy Valid:        \'" + proxy.isValid() + "\'");
         print("Proxy Remote:        \'" + proxy.isRemote() + "\'");
         print("Remote Proxy Valid: \'" + remoteProxy.isValid() + "\'");
         print("Remote Proxy Remote: \'" + remoteProxy.isRemote() + "\'");
      } catch (Exception e) {
         print("Failed to create and dispose of the child.");
         print(e.getMessage());
      }
   }
   static public String NAME = "ProxyDispatchExample";
   private void print(String s) { System.out.println(NAME + " (parent): " + s); }
   private static long SLEEP = 3000;
   private void pause() {
      try {
         Thread.sleep(SLEEP);
      } catch (InterruptedException ie) { }
   }
}
```

ProxyDispatchChild.java

```
package aglets.agletbook.chapter7;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class ProxyDispatchChild extends Aglet {
```

```
   public void onDisposing() {
      print("Being disposed of...");
   }
   public void run() {
      print("Running...");
   }
   private void print(String s) { System.out.println(ProxyDispatchExample.NAME + "
(child): " + s); }
}
```

The interesting point here concerns the outputs of the calls to isValid() and isRemote().

- **Before dispatching.** The child proxy is valid but not remote.
- **After dispatching.** The child proxy (called remoteProxy) is remote all right, but not valid. The aglet book on p. 171 claims that the remote proxy is valid. This would be true in version 1.0 but not version 1.1 of the aglet API.

New aglet methods

In the AgletContext class

AgletProxy CreateAglet(URL, String, Object)

- This is one of the most important methods. In most aglet systems, a stationary aglet, often called the "master" is used to create mobile aglets which are then dispatched to do their jobs. (Rather than using the Create button on Tahiti.)
- The URL argument is the location of the created aglet's class file. This can be null in which case the created aglet's code is with the same host as the aglet which calls create().
- The String argument is the name of the created aglet's class. The name is relative to the codebase of the aglet context, and must include package names as shown above (if there are any).
- The Object argument is an object passed to the on Creation(Object) method of the created aglet.

URL getHostingURL()

- Gets the URL of the server serving this aglet context.

In the AgletProxy class

AgletProxy dispatch(URL)

- Sends the aglet represented by this proxy to a remote host. Returns the remote proxy of the sent aglet. However it seems that the this remote proxy is invalid in version 1.1b1 of the aglet API. Note also that there is also a method void dispatch(URL) belonging to the Aglet class which allows an aglet to dispatch itself to another location.

boolean isRemote()
Checks if a proxy is remote.

boolean isValid()
Checks if the proxy is usable.

Remote Messaging Example

Often an remote aglet wants to send information back to its point of origin without returning there. The following example shows one way to do this using remote messaging. (You could also have the remote aglet itself return to its origin with the information.) The example uses yet another version of our friends, SayItAglet and HearItAglet, with the addition of a new "master" aglet.

There are 3 aglets in this example.

- HearItAglet4. Sends the message "Who are you?" to the remote SayItAglet4 aglet after it has received a "I have arrived safely." message from the SayItAglet4.

- SayItAglet4. The remote aglet. Sends a message "I have arrived safely." when it arrives at a remote host. Replies to the "Who are you?" message from the HearItAglet4.
- Master4. Creates HearItAglet4 and SayItAglet4 and dispatches the latter to the remote Tahiti server.

Master4.java

```
package aglets.mystuff.testtalk;
import com.ibm.aglet.*;
import java.net.*;

public class Master4 extends Aglet {
        private AgletContext thisContext = null;
        private AgletProxy sayItAgletRemoteProxy = null;
        private AgletProxy sayItAgletLocalProxy = null;
        private AgletProxy hearItAgletLocalProxy = null;

        public void onCreation(Object init) {
                try {
                        thisContext = getAgletContext();
                        hearItAgletLocalProxy = thisContext.createAglet(
                                                                null,
                                                                "aglets.mystuff.testtalk.HearItAglet4",
                                                                null);
                        sayItAgletLocalProxy = thisContext.createAglet(
                                                                null,
                                                                "aglets.mystuff.testtalk.SayItAglet4",
                                                                hearItAgletLocalProxy);
                        sayItAgletRemoteProxy = sayItAgletLocalProxy.dispatch(new URL("atp://localhost:9000"));
                }
                catch (Exception e) {
                        System.err.println("various possible exceptions");
                }
        }
}
```

SayItAglet4.java

```
package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class SayItAglet4 extends Aglet implements MobilityListener {

   private String [] msg  = new String[2];
   private AgletContext remoteContext;
   private AgletProxy aHomeProxy;

   public void onCreation(Object init) {
```

```
      msg[0] = new String("David Grimshaw");
      msg[1] =  new String(".. from SayItAglet4");
      aHomeProxy = (AgletProxy) init;
      addMobilityListener(this);
   }

   public String [] getmsg() {
      return msg;
   }
   public boolean handleMessage (Message msg) {
      if(msg.sameKind("Who are you?") ){
         msg.sendReply(getmsg());
         print("Message: Who are you? received.");
         return true;
      }
      return false;
   }
   public void onArrival (MobilityEvent mev) {
      remoteContext = getAgletContext();
      AgletID id = getAgletID();
      try {
         aHomeProxy.sendOnewayMessage(new Message("I have arrived safely",
remoteContext.getAgletProxy(id));
      }
      catch (Exception e) {
         System.err.println("send one way fails");
      }
   }
   public void onDispatching(MobilityEvent mev) {
   }
   public void onReverting(MobilityEvent mev) {
   }
   public static final String NAME = "SayItAglet4";
   public void print(String s) {
      System.out.println(NAME + ": " + s);
   }
}
```

Note.

This program implements a MobilityListener in the simplest way. Note that all three mobility listener methods must be implemented even though only one onArrival() does anything.

What's new

In the MobilityListener interface

- **void onArrival(MobilityEvent)**
  Executes on arriving at a destination. Completes before the run() method is restarted.
- **void onDispatching(MobilityEvent)**
  Executes just before the aglet leaves for a remote destination.
- **void onReverting(MobilityEvent)**
  Executes just after a remote aglet is told to revert to its place of origin.

In the AgletContext class

void SendOnewayMessage(String, Object|primitve)
Sends a message to which no reply is expected.

HearItAglet4.java

```
package aglets.mystuff.testtalk;
import com.ibm.aglet.*;
import java.util.*;

public class HearItAglet4 extends Aglet {

   AgletProxy ap = null;

   public boolean handleMessage(Message msg) {
      String [] ans = new String[2];
      ans[0] = "no answer";
      ans[1] = "no answer";
      if(msg.sameKind("I have arrived safely")) {
         try {
            ap = (AgletProxy) msg.getArg();
            ans = (String []) ap.sendMessage(new Message("Who are you?"));
         }
         catch(Exception e) {
            e.printStackTrace();
         }
         setText(ans[0] + " " + ans[1]);
         return true;
      }
      return false;
   }
}
```

Message Passing in the example.

The problem of knowing remote proxies is solved as follows.

- When created by Master4, SayItAglet4 is given an argument, **hearItAgletLocalProxy**, which reappears as the argument in its onCreation() method. It saves this reference as one of its members, **aHomeProxy**.
- Using its knowledge of the home (origin) proxy of HearItAglet4, SayItAglet4 can send HearItAglet4 the message, **I have arrived safely,** with a mesaage argument containing a reference to its proxy on its remote host.
- HearItAglet4 receives the message and gets SayItAglet4's remote proxy as the message argument. It uses this proxy to send SayItAglet4 the **Who are you** the message.
- Finally, SayItAglet4 uses the **Who are you** message object to send its final reply back to HearItAglet4.

More remote aglet examples for Oshima and Lange.

```
/*
 * @(#)ProxyDispatchExample.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter7;

import com.ibm.aglet.*;
import java.net.URL;

public class ProxyDispatchExample extends Aglet {

   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),

"aglets.agletbook.chapter7.ProxyDispatchChild",
                                                          null);
         print("Finished creating the child.");
                        pause();
                        print("Proxy Valid:        \'" + proxy.isValid() + "\'");
                        print("Proxy Remote:        \'" + proxy.isRemote() + "\'");
                        print("Dispatching the child...");
         String host = getAgletContext().getHostingURL().toString();
         URL destination = new URL("atp://localhost:9000");
         print("Destination is \'" + destination.toString() + "\'");
         AgletProxy remoteProxy = proxy.dispatch(destination);
                        print("Finished dispatching the child.");
                        pause();
                        print("Proxy Valid:        \'" + proxy.isValid() + "\'");
                        print("Proxy Remote:        \'" + proxy.isRemote() + "\'");
                        print("Remote Proxy Valid: \'" + remoteProxy.isValid() + "\'");
                        print("Remote Proxy Remote: \'" + remoteProxy.isRemote() + "\'");
      } catch (Exception e) {
         print("Failed to create and dispose of the child.");
         print(e.getMessage());
      }
   }

   static public String NAME = "ProxyDispatchExample";
   private void print(String s) { System.out.println(NAME + " (parent): " + s); }
   private static long SLEEP = 3000;
   private void pause() {
                try {
                        Thread.sleep(SLEEP);
```

```
                } catch (InterruptedException ie) { }
        }
}
```

```
/*
 * @(#)ProxyDispatchChild.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter7;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class ProxyDispatchChild extends Aglet {

   public void onDisposing() {
      print("Being disposed of...");
   }

   public void run() {
      print("Running...");
   }

   private void print(String s) { System.out.println(ProxyDispatchExample.NAME + "
(child): " + s); }
}
```

```
package aglets.mystuff.testtalk;

import com.ibm.aglet.*;

import java.net.*;

public class Master4 extends Aglet {

        private AgletContext thisContext = null;
        private AgletProxy sayItAgletRemoteProxy = null;
        private AgletProxy sayItAgletLocalProxy = null;
        private AgletProxy hearItAgletLocalProxy = null;

        public void onCreation(Object init) {
                try {
                        thisContext = getAgletContext();
                        hearItAgletLocalProxy = thisContext.createAglet(null,
"aglets.mystuff.testtalk.HearItAglet4", sayItAgletRemoteProxy);
                        sayItAgletLocalProxy = thisContext.createAglet(null,
"aglets.mystuff.testtalk.SayItAglet4", hearItAgletLocalProxy);
                        sayItAgletRemoteProxy = sayItAgletLocalProxy.dispatch(new
URL("atp://localhost:9000"));
                        // unfortunately this remote proxy is invalid in version 1.1b1.
The aglet book is based
                        // on version 1.02. See ch 7. p. 121. Running this
(aglets.agletbook.chapter7.
                        // ProxyDispatchExample verifies this. THe remote proxy returned
by dispatch() is
                        // invalid, contradicting what the book says.

                }
                catch (Exception e) {
                        System.err.println("various possible exceptions");
                }
        }

}
```

```
/**
 * sayItAglet3.java
 * Inter aglet communication.
 * This aglet is perpared to receive a "Who are you?" message and reply to
 * it.
 */

package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class SayItAglet4 extends Aglet implements MobilityListener {
        private String [] msg  = new String[2];
        private AgletContext remoteContext;
        private AgletProxy aHomeProxy;

        public void onCreation(Object init) {
                msg[0] = new String("David Grimshaw");
                msg[1] =  new String(".. from SayItAglet4");
                aHomeProxy = (AgletProxy) init;
                addMobilityListener(this);
        }



        public String [] getmsg() {
                return msg;
        }

        public boolean handleMessage (Message msg) {
                if(msg.sameKind("Who are you?") ){
                        msg.sendReply(getmsg());
                        print("Message: Who are you? received.");
                        return true;
                }
                return false;
        }

        public void onArrival (MobilityEvent mev) {
                remoteContext = getAgletContext();
                AgletID id = getAgletID();
                try {
                        aHomeProxy.sendOnewayMessage(new Message("I have arrived safely",
remoteContext.getAgletProxy(id)));
                }
                catch (Exception e) {
                        System.err.println("send one way fails");
                }
        }
        public void onDispatching(MobilityEvent mev) {
        }
        public void onReverting(MobilityEvent mev) {
        }

        public static final String NAME = "SayItAglet4";
        public void print(String s) {
                System.out.println(NAME + ": " + s);
        }

}
```

```
/**
 * HearItAglet4.java
 * This sends a message to SayItAGlet3
 * You don't need to check which aglets receive the message, since those
 * which do not understand the message do not reply.
 * On the other hand, this is a waste of energy so it is probably a good idea
 * to be selective anyway. (Below the selection method used in HearItAglet2 has been
 * commented out.
 * DG Sept. 99
 */

package aglets.mystuff.testtalk;

import com.ibm.aglet.*;
import java.util.*;

public class HearItAglet4 extends Aglet {
        AgletProxy ap = null;

        public boolean handleMessage(Message msg) {
                String [] ans = new String[2];
                ans[0] = "no answer";
                ans[1] = "no answer";
                if(msg.sameKind("I have arrived safely")) {
                        try {
                                ap = (AgletProxy) msg.getArg();
                                ans = (String []) ap.sendMessage(new Message("Who are
you?"));
                        }
                        catch(Exception e) {
                                e.printStackTrace();
                        }
                        setText(ans[0] + " " + ans[1]);
                        return true;
                }
                return false;
        }

}
```

Remote Communication Examples from Aglet Book

A Remote Messaging example from Chpater 6 of Lange & Oshima

ListingAglet1.java

This is the mobile aglet which obtains a directory listing from a remote host and returns it to its master using messaging.

```
package aglets.agletbook.chapter6;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.io.*;

public class ListingAglet1 extends Aglet {

   AgletProxy _proxy = null;
   File _dir = null;

   public void onCreation(Object args) {

      _dir = (File)((Object[])args)[0];
      _proxy = (AgletProxy)((Object[])args)[1];

      addMobilityListener(
         new MobilityAdapter() {

            public void onArrival(MobilityEvent me) {
                try {
                  _proxy.sendMessage(new Message("Arrived", "I've arrived."));
                  _proxy.sendMessage(new Message("Listing", _dir.list()));
                } catch (Exception e) {
                   dispose();
                }
             }
         }
      );
   }
}
```

Note how the arguement to onCreation() is handled. This is how the master's proxy (and other information) is passed to the slave.

ListingAgletMaster1.java

Here is the master aglet which creates the slave and passes information to it. Particularly important is to let the slave know the master's proxy.

```
package aglets.agletbook.chapter6;
import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;
import java.io.*;
```

```
public class ListingAgletMaster1 extends Aglet {
   public void run() {
      print("STARTING --------------------");
      try {
          URL destination = new URL("atp://proton.scs.ryerson.ca:4434");
          Object[] args = new Object[] { new
File("I:\\coursesf99\\cps720\\assignment1"),
getAgletContext().getAgletProxy(getAgletID()) };
          AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),
"aglets.agletbook.chapter6.ListingAglet1", args);
          proxy = proxy.dispatch(destination);
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }
   public boolean handleMessage(Message msg) {
      print("is handling a message...");
      if (msg.sameKind("Listing")) {
         String[] list = (String[])msg.getArg();
         for (int i = 0; i < list.length; )
            System.out.println(i + ": " + list[i++]);
         return true;     // Yes, I handled this message.
      } else if(msg.sameKind("Arrived")) {
            System.out.println("Slave reports arrival at destination");
            return true;
      } else
         return false;    // No, I did not handle this message.
   }
   static public String NAME = "ListingAgletMaster1";
   private void print(String s) { System.out.println(NAME + ": " + s); }
   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch (InterruptedException
ie) { } }
}
```

The directory listing is returned to the master which outputs it to stdout.

Using an aglet to carry data

Instead of messaging, the slave can itself return from the remote site with the collected data. Although this method is a bit "heavier" than messaging, it has its advantages . For example, if the remote aglet fails to dispatch itself back home for some reason, the home base can still try to retrieve it, and its cargo, using Tahiti's retract method. Or you might leave the aglet 'parked' remotely while you disconnect, retrieving the aglet later when you reconnect.

ListingAglet2.java

This is a simple aglet which dispatches itself to a remote site and then (attempts) to dispatch itself back to its origin. Note the use of a flag to distinguish arrivals at the remote site and at home base.

```
package aglets.agletbook.chapter6;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.io.*;
import java.net.*;
```

```
public class ListingAglet2 extends Aglet {
   boolean back = false;
   String[] list = null;
   URL origin = null;
   File dir = new File("/home/dgrimsha/aglets");
   public void onCreation(Object o) {
      addMobilityListener(
         new MobilityAdapter() {
            public void onArrival(MobilityEvent me) {
               if (back) {
                  for (int i = 0; i < list.length; )
                     System.out.println(i + ": " + list[i++]);
                  dispose();
               } else {
                  try {
                    list = dir.list();
                    back = true;
                    dispatch(origin);
                  } catch (Exception e) {
                     dispose();
                  }
               }
            }
         }
      );
      origin = getAgletContext().getHostingURL();
      try {
          dispatch(new URL("atp://jupiter.scs.ryerson.ca:12001"));
      } catch (Exception e) {
      }
   }
}
```

Future Replies

Asynchronous messages are supported by aglets. This allows, for example, a master aglet to continue with its work without waiting for its slave to finish its task. In other words, the master aglet does not block while waiting for a message from the slave.

An example from Chapter 6 of Oshima and Lange.

Theparent.

FutureExample.java

```
package aglets.agletbook.chapter6;

import com.ibm.aglet.*;

public class FutureExample extends Aglet {
   static private int Future = 4;
   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),

"aglets.agletbook.chapter6.FutureChild",
                                                          null);
         print("Finished creating the child.");
         pause();
         try {
            print("Sends a message...");
            // you could also use FutureReply sendFutureMessage(Message)
            FutureReply future = proxy.sendAsyncMessage(new Message("Please reply"));
            print("The message was sent.");
            while (!future.isAvailable())              // Checks whether a reply is
available.
               doIncrement();                          // If not: do incremental
work...
            String reply = (String)future.getReply();   // Gets the reply.
            print("Got the reply: \'" + reply + "\'");
         } catch (NotHandledException e) {
            print("Failed to send the message.");
            print(e.getMessage());
         }
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }
   void doIncrement() {
      print("Working...");
      shortPause();
   }
   public static  String NAME = "Future";
   private void print(String s) {
```

```
      System.out.println(NAME + " (parent): " + s);
   }
   private static long SLEEP = 3000;
   private void pause() {
      try {
         Thread.sleep(SLEEP);
      }
      catch (InterruptedException ie) { } }
   private void shortPause() {
      try {
         Thread.sleep(SLEEP/4);
      } catch (InterruptedException ie) { } }
}
```

The child.

FutureChild.java

```
package aglets.agletbook.chapter6;

import com.ibm.aglet.*;

public class FutureChild extends Aglet {
   public void run() {
      print("Running...");
   }
   public boolean handleMessage(Message msg) {
      print("Received a message: \'" + msg.getKind() + "\'");
      if (msg.sameKind("Please reply")) {
         print("Handling \'Please reply\'.");
         pause(); pause();
         print("Replying...");
         msg.sendReply("Hello World!");
         return true;
      }
      return false;
   }
   private static long SLEEP = 3000;
   private void pause() {
      try {
         Thread.sleep(SLEEP);
      } catch (InterruptedException ie) { } }
   private void print(String s) {
      System.out.println(FutureExample.NAME + "  (child): " + s);
   }
}
```

On the home console, the message "Working ..." is printed a number of times before the message "Hello World" is received from the child and also printed.

```
/*
 * @(#)FutureExample.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter6;

import com.ibm.aglet.*;

public class FutureExample extends Aglet {

   static private int Future = 4;

   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),

"aglets.agletbook.chapter6.FutureChild",
                                                          null);
         print("Finished creating the child.");
         pause();
         try {
            print("Sends a message...");
            FutureReply future = proxy.sendAsyncMessage(new Message("Please reply"));
            print("The message was sent.");
            while (!future.isAvailable())              // Checks whether a reply is
available.
               doIncrement();                          // If not: do incremental work...
            String reply = (String)future.getReply();  // Gets the reply.
            print("Got the reply: \'" + reply + "\'");
         } catch (NotHandledException e) {
            print("Failed to send the message.");
            print(e.getMessage());
         }
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }

   void doIncrement() {
      print("Working...");
      shortPause();
   }
```

```
  public static  String NAME = "Future";
  private void print(String s) {
     System.out.println(NAME + " (parent): " + s);
  }
  private static long SLEEP = 3000;
  private void pause() {
     try {
        Thread.sleep(SLEEP);
     }
     catch (InterruptedException ie) { } }
  private void shortPause() {
     try {
        Thread.sleep(SLEEP/4);
     } catch (InterruptedException ie) { } }
}
```

```
/*
 * @(#)FutureChild.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter6;

import com.ibm.aglet.*;

public class FutureChild extends Aglet {

   public void run() {
      print("Running...");
   }

   public boolean handleMessage(Message msg) {
      print("Received a message: \'" + msg.getKind() + "\'");
      if (msg.sameKind("Please reply")) {
         print("Handling \'Please reply\'.");
         pause(); pause();
         print("Replying...");
         msg.sendReply("Hello World!");
         return true;
      }
      return false;
   }

   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch (InterruptedException ie) {
} }
   private void print(String s) { System.out.println(FutureExample.NAME + "  (child): " +
s); }
}
```

Aglet Programming Basics

Aglet programming revolves around two classes and two interfaces.

- Aglet
- *AgletContext*
- *AgletProxy*
- Message

The AgletProxy is intended to shield aglets proper from outside scrutiny. For example, communication via messages is handled by the aglet proxies.

The AgletContext provides an environment for the aglets. For example, the Tahiti server provides such an environment. This environment is closed off from the host operating system (a "sandbox" as for applets). Thus the host is protected from malicious aglets.

Basic Example

To introduce the most important concepts, consider this example. There are two aglets, BasicMaster and BasicChild. The user sets up two Tahiti servers. (In the example, one at default port 4434 and the other at 9000. On the 4434 Tahiti, the user creates BaseMaster. In turn, BaseMaster creates BaseChild and dispatches it to the 9000 Tahiti. On arriving, the child sends a messsage to the master saying it has arrived. The master responds to this message by retracting the child back to the 4434 Tahiti. The master does not move.

Here is BasicMaster. Important Aglet items are shown in bold red.

```
/** BasicMaster.java
*
*/
package aglets.mystuff.basic;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;

public class BasicMaster extends Aglet {
        URL target;
        AgletID childID;
        public void run() {
                try {
```

```
                target= new URL("atp://notebook:9000");
                AgletProxy masterProxy = getProxy();
                AgletProxy childProxy = getAgletContext().createAglet(null,
                        "aglets.mystuff.basic.BasicChild",
                        masterProxy);
                childID = childProxy.getAgletID();
                childProxy.dispatch(target);
            } catch (Exception e) {}
        }
        public boolean handleMessage(Message msg) {
            if(msg.sameKind("arrived")) {
                System.out.println("Received message");
                try {
                    getAgletContext().retractAglet(target, childID);
                } catch (Exception e) {
                    System.err.println(e.toString());
                }
                return true;
            }
            return false;
        }
    }
```

Notes on BasicMaster

- Aglets can find out where they are using the method getContext() of the Aglet class.
- The method createAglet() belongs to the AgletContext class. It takes three arguments:
  - The codebase of the aglet beiing created. Use null for aglets created from local code, that is, classes in a subldirectory of AGLET_PATH. (or, see aglets.class.path entry in the .props file).
  - The fully qualified name of the aglet class (without the .class extension) as a string.
  - A reference to an Object which the aglet system will pass as argument ot the created aglet's onCreation() method. Null if you do not wish to pass anything to the child.
- The child is given the master's proxy so it can send the master messages from a remote location.
- Note that the master aglet remembers the child's ID for use in the retractAglet() method.
- The handleMessage() ands sameKind() methods are the standard way of receiving messages.

Here is the mobile agent BasicChild.

```
/** BasicChild.java
```

```
*
*/

package aglets.mystuff.basic;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class BasicChild extends Aglet {

    private boolean atHome;
    private AgletProxy masterProxy;

    public void onCreation (Object init) {
        atHome = true;
        masterProxy = (AgletProxy) init;
        setText("Alive!");
        addMobilityListener(new MobilityAdapter () {
            public void onArrival(MobilityEvent me) {
                setText("Arrived safely");
                if(atHome) atHome = false;
                else atHome = true;
            }
            public void onReverting(MobilityEvent mev) {
                System.out.println("Goodbye");
            }
        });
    }
    public void run() {
        if(!atHome) {
            try {
                Thread.sleep(2000);
                // NB! one way message. If you use sendMessage() child aglet
                // is crashed by the the reply which comes while it is trying to revert.
                masterProxy.sendOnewayMessage(new Message("arrived"));
            } catch (Exception e) {
                System.err.println(e.toString());
            }
```

```
                } else {
                        setText("Home, sweet home.");
                }
        }
}
```

Notes on BasicChild

- To send a message back to the aglet that created it, BasicChild needs to know the proxy of its master. This is passed to it via the argument to the important message onCreation(). (onCreation() belongs to the Aglet class.)
- run() is available to all aglets because they run on separate threads. run() is executed after onArrival() if onArrival() exists. It is always executed "from the top" at each stop the aglet makes because Java cannot save the complete state of the computation (i.e., the instruction pointer cannot be saved).
- You need some kind of switch to distinguish arrivals at a remote host from arrivals "back home". The if(athome) ... statement, above, takes care of this in this example.
- Messages are most often sent with the sendMessage() method because a reply is usually expected. In this case, sendOnewayMessage() is used because a reply is not expected, and, in fact, causes a crash.

[top] [previous] [next] Questions?

```
/** BasicMaster.java
 *
 */
 package aglets.mystuff.basic;

 import com.ibm.aglet.*;
 import com.ibm.aglet.event.*;
 import java.net.URL;

 public class BasicMaster extends Aglet {
      URL target;
     AgletID childID;
     public void run() {
         try {
             target= new URL("atp://notebook:9000");
             AgletProxy masterProxy = getProxy();
             AgletProxy childProxy = getAgletContext().createAglet(null,

"aglets.mystuff.basic.BasicChild",

masterProxy);
             childID = childProxy.getAgletID();
             childProxy.dispatch(target);
         } catch (Exception e) {}
     }
     public boolean handleMessage(Message msg) {
         if(msg.sameKind("arrived")) {
             System.out.println("Received message");
             try {
                getAgletContext().retractAglet(target, childID);
            } catch (Exception e) {
                System.err.println(e.toString());
            }
             return true;
         }
         return false;
     }
 }
```

```
/** BasicChilld.java
*
 */

  package aglets.mystuff.basic;

  import com.ibm.aglet.*;
  import com.ibm.aglet.event.*;
 import java.net.URL;

 public class BasicChild extends Aglet {

     private boolean atHome;
     private AgletProxy masterProxy;

     public void onCreation (Object init) {
         atHome = true;
         masterProxy = (AgletProxy) init;
         setText("Alive!");
         addMobilityListener(new MobilityAdapter () {
             public void onArrival(MobilityEvent me) {
                 setText("Arrived safely");
                 if(atHome) atHome = false;
                 else atHome = true;
             }
             public void onReverting(MobilityEvent mev) {
                 System.out.println("Goodbye");
             }
         });
     }
     public void run() {
          if(!atHome) {
              try {
                  Thread.sleep(2000);
                 // NB! one way message. If you use sendMessage() child aglet
                 // is crashed by the the reply which comes while it is trying to revert
                 masterProxy.sendOnewayMessage(new Message("arrived"));
             } catch (Exception e) {
                 System.err.println(e.toString());
             }
          } else {
              setText("Home, sweet home.");
          }
      }
 }
```

An Aglet with a separate itinerary class

Most of the examples so far do not separate controlling the Aglet's itinerary from the tasks it carries out. This lack of modularity could be a problem for more complex problems. Correct coding and code maintenance could be a problem. The example shown below illustrates using a separate class to to control the Aglet's itinerary.

Boomerang Example from Lange & Oshima (p. 40-42)

(with some corrections and additions.)

The boomerang aglet is simple. It uses an itinerary class to indirectly implement a listener for mobility events. This itinerary class is "plugged into" the aglet. This design pattern will come up on several later occasions as well.

BoomerangAglet.java

```
package aglets.mystuff.boomerang;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class BoomerangAglet extends Aglet {
        private String remote = "atp://localhost:9000";

        public void onCreation(Object init) {
                addMobilityListener(new BoomerangItinerary(this));
        }
}
```

BoomerangItinerary.java

```
package aglets.mystuff.boomerang;

import com.ibm.aglet.*;
```

```
import com.ibm.aglet.event.*;

import java.net.URL;

public class BoomerangItinerary implements MobilityListener {
        private Aglet target = null;
        private String origin = null;

        public BoomerangItinerary (Aglet target) {
                this.target = target;
                target.addMobilityListener(this);
                origin = target.getAgletInfo().getOrigin();
                System.out.println(origin);
        }

        public void onDispatching(MobilityEvent mev) {
                target.setText("I'm leaving for " + mev.getLocation());
        }

        public void onArrival(MobilityEvent mev) {

                if(atOrigin(mev.getLocation()) == false) {
                        // this test always returns false if the
                        // sender is the default port 4434 !?
                        // Bug in beta 1.1 ?
                        System.out.println(mev.getLocation().toString());
                        try {
                                target.dispatch(new URL(origin));
                        }
                        catch (Exception e) {}
                }
        }

        public void onReverting(MobilityEvent mev) {
        }

        public boolean atOrigin(URL current) {
                return origin.equals(current.toString());
```

```
        }
    }
```

[top] [previous] [next] Questions?

```
package aglets.mystuff.boomerang;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class BoomerangAglet extends Aglet {
        private String remote = "atp://localhost:9000";

        public void onCreation(Object init) {
                addMobilityListener(new BoomerangItinerary(this));
        }
}
```

```
package aglets.mystuff.boomerang;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

import java.net.URL;

public class BoomerangItinerary implements MobilityListener {
        private Aglet  target = null;
        private String origin = null;

        public BoomerangItinerary (Aglet target) {
                this.target  = target;
                target.addMobilityListener(this);
                origin = target.getAgletInfo().getOrigin();
                System.out.println(origin);
        }

        public void onDispatching(MobilityEvent mev) {
                target.setText("I'm leaving for "  + mev.getLocation());
        }

        public void onArrival(MobilityEvent mev) {

         if(atOrigin(mev.getLocation()) == false) {
                         // this test always returns false if the
                         // sender is the default port 4434 !?
                         // Bug in beta 1.1 ?
                        System.out.println(mev.getLocation().toString());
                        try {
                                target.dispatch(new URL(origin));
                        }
                        catch (Exception e) {}
                }
        }

        public void onReverting(MobilityEvent mev) {
        }

        public boolean atOrigin(URL current) {
                return origin.equals(current.toString());
        }
}
```

Aglet Cloning

The Aglet API provides a cloning facility for Aglets. This is actually a second way to create new Aglets - by making copies of already existing Aglets. Cloning can be useful if parallel processing is needed.

The only diffrence between an Aglet and its clone is that they have different AgletID's (which means that their proxies are not quite the same).

Creating clones

There are various clone() methods to do this.

An Aglet clones itself

public final Object clone() throws CloneNotSupportedException

A clone of an Aglet can be created using its proxy.

public Object clone() throws CloneNotSupportedException

The Object returned is the clone's proxy.

The Clone Event and Listener

Cloning generates an event. To make use of this you have to implement a CloneListener. This is usually done via a CloneAdapter class, mostly as an inner class.

There are three callback methods assoicated with the CloneListener.

public void onCloning(CloneEvent e)

This is used for clone initialization. It is called in the original Aglet's thread.

public void onClone(CloneEvent e)

This is also used to initialize the clone. It is called after onCloning(). In particular, it is used to set a flag to distinguish clone code from the original Aglet's code (see example below).

public void onCloned(CloneEvent e)

This method is called in the clone's thread after it has been created. After it returns, the clone's run() method is invoked. (onArrival() plays an analagous role for MobilityEvents.)

A Cloning Example

This example prints out the names of 7 clones created from one original. Notice how the clone code is separated from that of the original. All 8 Aglets have their own threads.

CloneExampleAglet.java

The code:

```
package aglets.cloneexample;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

import java.util.*;

/**
* Illustrates cloning. 7 clones are created and survive 2 seconds.
*/
public class CloneExampleAglet extends Aglet {
    boolean isClone = false;
    private Vector dwarfs;
    private String dwarfName = null;

    public void onCreation(Object init) {
        dwarfs = new Vector();
        dwarfs.addElement("sleepy");
        dwarfs.addElement("dopy");
```

```
        dwarfs.addElement("grumpy");
        dwarfs.addElement("sneezy");
        dwarfs.addElement("happy");
        dwarfs.addElement("bashful");
        dwarfs.addElement("doc");

        addCloneListener(new CloneAdapter() {
            public void onClone(CloneEvent ce) {
                isClone = true;
            }
        } );
    }
    public void run() {
        if(!isClone) {
            for(Enumeration e = dwarfs.elements();
            e.hasMoreElements();) {
                dwarfName = (String) e.nextElement();
                try {
                    clone();
                } catch(Exception ex) {
                    System.err.println(ex);
                }
```

```
            }
        } else {
            System.out.println("HiHo, HiHo .. I'm " + dwarfName);
            setText("HiHo, HiHo .. I'm " + dwarfName);
            pause(2000);
            dispose();
        }
    }

    private void pause(int time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {}
    }
}
```

[top] [previous] [next] Questions?

```
package aglets.cloneexample;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

import java.util.*;

/**
* Illustrates cloning. 7 clones are created and survive 2 seconds.
*/
public class CloneExampleAglet extends Aglet {
        boolean isClone = false;
        private Vector dwarfs;
        private String dwarfName = null;

        public void onCreation(Object init) {
                dwarfs = new Vector();
                dwarfs.addElement("sleepy");
                dwarfs.addElement("dopy");
                dwarfs.addElement("grumpy");
                dwarfs.addElement("sneezy");
                dwarfs.addElement("happy");
                dwarfs.addElement("bashful");
                dwarfs.addElement("doc");

                addCloneListener(new CloneAdapter() {
                        public void onClone(CloneEvent ce) {
                                isClone = true;
                        }
                } );
        }
        public void run() {
                if(!isClone) {
                        for(Enumeration e = dwarfs.elements(); e.hasMoreElements();) {
                                dwarfName = (String) e.nextElement();
                                try {
                                        clone();
                                } catch(Exception ex) {
                                        System.err.println(ex);
                                }
                        }
                } else {
                        System.out.println("HiHo, HiHo .. I'm " + dwarfName);
                        setText("HiHo, HiHo .. I'm " + dwarfName);
                        pause(2000);
                        dispose();
                }
        }

        private void pause(int time) {
                try {
                        Thread.sleep(time);
                } catch (InterruptedException e) {}
        }
}
```

Survey of Aglet Design Patterns

[See Chapter 8 of Oshima and Lange's Aglet Book.]

Design patterns have become important in object oriented programming. (Design Patterns vs Universal Modeling Language ?!) Chapter 8 of the aglet book discusses several mobile agent design patterns in general, and two in particular.

Aglet Patterns

TABLE 8-1 Agent Design Patterns

Pattern	*Description*
TRAVELING PATTERNS	
Itinerary	Objectifies aglets' itineraries and routing among destinations.
Forwarding	Provides a way for a host to forward newly arrived aglets automatically to another host.
Ticket	Objectifies a destination address and encapsulates the quality of service and permissions needed to dispatch an aglet to a host address and execute it there.
TASK PATTERNS	
Master-Slave	Defines a scheme whereby a master aglet can delegate a task to a slave aglet.
Plan	Provides a way of defining the coordination of multiple tasks to be performed on multiple hosts.
INTERACTION PATTERNS	
Meeting	Provides a way for two or more aglets to initiate local interaction at a given host.
Locker	Defines a private storage space for data left by an aglet before it is temporarily dispatched (sent) to another destination.
Messenger	Defines a surrogate aglet to carry a remote message from one aglet to another.
Finder	Defines an aglet that provides services for naming and locating aglets with specific capabilities.
Organized Group	Composes aglets into groups in which all members of a group travel together.

The master-slave and itinerary patterns are discussed in detail.

Aglet API patterns

The Aglet API has a number of patterns built in. (Some of them have bugs.) These can be fouund in the com.ibm.agletx.utils and com.ibm.agletx.patterns. There is some documentaion as well..

Master-Slave Pattern

This is the simplest design pattern. Design patterns are discussed under the following headings.

- Definition
- Purpose
- Applicability
- Participants
- Collaboration
- Consequences
- Implementation

Definition

A pattern where a master can delegate a task to a slave.

Purpose

A task is split between two computers. A parallel process is possible. While the slave is away doing its task, the master can continue with its task. The slave usually sends the result of its taks back to the master.

Applicability

parallleism is needed

the master needs to get something done on another machine

Participants

- Abstract slave
- Concrete slave
- Master

Collaboration

The master-slave participants cooperate as follows:

1. A master aglet creates a slave aglet
2. The slave initializes its task
3. The slave moves to a remote host and executes its task.
4. The slave sends the result of its task to the master
5. The slave disposes itself

Consequences

The constant parts of the design are separated from the variable parts. The constant parts need only be implemented once, and developers can concentrate on the variable point.

Implementation

(Aglet book, chapter 8)

The implementation is built aound a foundation abstract class: (called Slave.java in the textbook).

Slave1.java

```
package aglets.agletbook.chapter8;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public abstract class Slave1 extends Aglet {
   URL destination = null;
   AgletProxy master = null;
   public void onCreation(Object args) {
      try {
         destination = (URL)((Object[])args)[0];
         master = (AgletProxy)((Object[])args)[1];
         initializeTask();
         addMobilityListener(
            new MobilityAdapter() {
               public void onArrival(MobilityEvent me) {
                 print("Arrived...");
                 try {
                     master.sendMessage(new Message("Result",
doTask()));
                     dispose();
                 } catch (Exception e) {
                    print("Failed to send result to master.");
                    print(e.getMessage());
                 }
               }
            }
         );
         dispatch(destination);
      } catch (Exception e) {
         print("Failed to create slave.");
         print(e.getMessage());
      }
   }
   abstract void initializeTask();
   abstract Object doTask();
```

```
   static public String NAME = "Slave1";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

The slave's mobility action on arrival is always the same, so we abstract it out in this abstract class. The same is true of the slave's messaging action.

What is different for each slave is (1) how it is initialzed, and (2) what its task is. The corresponding methods initializeTask() and doTask() are implemented in a subclass of this one.

Lange and Oshima provide this simple example. The abstraction is made concrete.

In this example, the task is just to print a message to stdout on the receiver, and send a string back to the master aglet at home.

MySlave.java

```
package aglets.agletbook.chapter8.MySlave;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public class MySlave extends Slave1 {
   public void initializeTask() {
      print("Initializing.");
   }
   public Object doTask() {
      print("Performs task");
      return "Some result...";
   }
   static public String NAME = "MySlave";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

A master aglet at home creates the concrete slave and sends it off to do its thing.

MyMaster.java

```
package aglets.agletbook.chapter8;
import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;

public class MyMaster extends Aglet {
   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         String host =
getAgletContext().getHostingURL().toString();
```

```
         URL destination = new
URL("atp://proton.scs.ryerson.ca:9000");
         AgletProxy thisProxy =
getAgletContext().getAgletProxy(getAgletID());
         Object[] args = new Object[] { destination, thisProxy };
         getAgletContext().createAglet(getCodeBase(),

"aglets.agletbook.chapter8.MySlave",
                                       args);
         print("Finished creating the child.");
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }
   public boolean handleMessage(Message msg) {
      if (msg.sameKind("Result"))
         print("Received a result: \'" + msg.getArg() + "\'");
      return true;
   }
   static public String NAME = "MyMaster";
   private void print(String s) { System.out.println(NAME + ": " +
s); }
   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch
(InterruptedException ie) { } }
}
```

A typical ouput on the home server would be:

MYMASTER: STARTING ----------

MYMASTER: Creating the child ...

MYMASTER: Finished createing the child.

MySlave: Intitializing ...

MyMaster: Received a result: 'Some result ...'

```
/*
 * @(#)Slave1.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public abstract class Slave1 extends Aglet {

//   SlaveSetup _setup = null;
   URL destination = null;
   AgletProxy master = null;

   public void onCreation(Object args) {
      try {
//       _setup = (SlaveSetup)init;
         destination = (URL)((Object[])args)[0];
         master = (AgletProxy)((Object[])args)[1];
         initializeTask();
         addMobilityListener(
            new MobilityAdapter() {
               public void onArrival(MobilityEvent me) {
                 print("Arrived...");
                 try {
                     master.sendMessage(new Message("Result", doTask()));
                     dispose();
                 } catch (Exception e) {
                    print("Failed to send result to master.");
                    print(e.getMessage());
                 }
                     }
            }
         );
         dispatch(destination);
      } catch (Exception e) {
         print("Failed to create slave.");
         print(e.getMessage());
      }
   }

   abstract void initializeTask();

   abstract Object doTask();
```

```
   static public String NAME = "Slave1";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

```
/*
 * @(#)MySlave.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public class MySlave extends Slave1 {

   public void initializeTask() {
      print("Initializing.");
   }

   public Object doTask() {
      print("Performs task");
      return "Some result...";
   }

   static public String NAME = "MySlave";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

```
/*
 * @(#)MyMaster.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8;

import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;

public class MyMaster extends Aglet {

   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         String host = getAgletContext().getHostingURL().toString();
         URL destination = new URL("atp://proton.scs.ryerson.ca:9000");
         AgletProxy thisProxy = getAgletContext().getAgletProxy(getAgletID());
//         SlaveSetup setup = new SlaveSetup(destination, thisProxy);
         Object[] args = new Object[] { destination, thisProxy };
         getAgletContext().createAglet(getCodeBase(),
                                       "aglets.agletbook.chapter8.MySlave",
                                       args);
         print("Finished creating the child.");
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }

   public boolean handleMessage(Message msg) {
      if (msg.sameKind("Result"))
         print("Received a result: \'" + msg.getArg() + "\'");
      return true;
   }

   static public String NAME = "MyMaster";
   private void print(String s) { System.out.println(NAME + ": " + s); }
   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch (InterruptedException ie) {
} }
}
```

Itinerary Pattern

(chapter 8)

Definition

This pattern objectifies trips of aglets to multiple destinations

Purpose

Responsibility for navigation is off loaded from the aglet itself to an associated itinerary object. This separation of the navigation control from the aglet proper increases flexibility. The same itinerary can be plugged into different aglets. Or aglets can be given different types of itinerary without having to modify the aglet's own code.

Applicability

Use this pattern when

- You want to hide the aglet travel plan from its behaviour in order to promote modularity.
- Provide a uniform travel interface for aglets
- Define travel plans which can be shared or reused

Participants

- Abstract Itinerary. Defines a model for an itinerary with two abstract methods, go() and hasMoreDestinations().
- ConcreteItinerary. Implements the abstract methods of the Itinerary class and keeps track of the current destination of the aglet.
- Aglet. The aglet following the itinerary.

The aglet and itinerary must be connected. The aglet creates the itinerary and initializes it with,

1. A list of destinations to be visited sequentially
2. A reference to the aglet.

The aglet uses the itinerary's go() method to dispatch itself, and its companion itinerary object.

Collaboration

- The ConcreteItinerary object is initialized by the aglet.
- The ConcreteItinerary object dispatches the aglet to the first destination.
- When the aglet invokes the itinerary's go() method, it is dispatched to the next destination.

Consequences

The pattern supports variations in navigation. For example, by changing the failure to dispatch exception handling, various behaviours can be generated when the destination is not available. Such changes will not require the aglet itself to be modified. Different aglets can also be made to share travel plans as embodied in itinerary objects.

Implementation

The three participants in this pattern are implemented in 3 files:

- Itinerary.java
- SeqIntinerary.java
- ItinerantAglet.java

To get things started, a stationary aglet is implemented in Parent.java

I have put these in a package aglets.agletbook.chapter8a.

Parent.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;
import java.util.*;

public class Parent extends Aglet {
   Itinerary _itinerary = null;
   public void onCreation(Object init) {
      try {
         Vector destinations = new Vector();
         destinations.addElement(new URL("atp://localhost:9000"));
         destinations.addElement(new URL("atp://localhost:9001"));
         destinations.addElement(new URL("atp://localhost:9002"));
         destinations.addElement(new URL("atp://localhost:9003"));
         destinations.addElement(new URL("atp://localhost:9004"));
         destinations.addElement(new URL("atp://localhost:9005"));
         URL origin = getAgletContext().getHostingURL();
         getAgletContext().createAglet(getCodeBase(),

"aglets.agletbook.chapter8a.ItinerantAglet",
                                       new SeqItinerary(origin,
destinations));
      } catch (Exception e) {
         print("Failed to initalize the itinerary.");
         print(e.getMessage());
      }
   }
   static public String NAME = "ParentItinerantAglet";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

Note how the travelling aglet, ItineraryAglet, is created knowing the home proxy of its parent, and carrying a list (Vector) of

its planned destinations.

The navigation requirements are encapsulated in the abstract Itinerary class.

Itinerary.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import java.net.*;
import java.io.*;

public abstract class Itinerary implements Serializable {

   protected URL _origin = null;
   protected AgletProxy _aglet = null;

   public Itinerary(URL origin) {
      _origin = origin;
   }
   public void init(Aglet aglet) {
      _aglet =
aglet.getAgletContext().getAgletProxy(aglet.getAgletID());
      go();
   }
   public URL getOrigin() {
      return _origin;
   }
   public AgletProxy getAglet() {
      return _aglet;
   }

   protected void go(URL destination) throws Exception {
      _aglet.dispatch(destination);
   }

   public abstract void go();
   public abstract boolean hasMoreDestinations();
   public abstract URL getNextDestination();
}
```

Notes.

- The name _aglet is unfortunate. This is an AgletProxy, not an Aglet.
- The class is abstract so it cannot be instantiated. To use it you must subclass it.
- The Itinerary goes along with its Aglet partner, so it must be serializable. Its subclasses inherit this property.

- Go is overloaded. The go(URL) just invokes dispatch to move the aglet to its next destination. The go() with no arguments is more general, and can be used recursively. It is implemented to control the whole journey.

SeqItinerary.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class SeqItinerary extends Itinerary {

   private Vector _destinations = null;

   SeqItinerary(URL origin, Vector destinations) {
      super(origin);
      _destinations = (Vector)destinations.clone();
   }

   public void go() {
      URL dest = getNextDestination();
      if (dest != null) {
         _destinations.removeElementAt(0);
         try {
            go(dest);
         } catch (Exception e) {
            print("Failed to dispatch (" + dest + ")");
            print("Skip this destination and go for the next.");
            print(e.getMessage());
            go(); // recursion!
         }
      }
   }

   public boolean hasMoreDestinations() {
      return _destinations.size() > 0;
   }

   public URL getNextDestination() {
      if (hasMoreDestinations())
         return (URL)_destinations.firstElement();
      else
         return null;
```

```
    }

    static public String NAME = "SeqItinerary";
    void print(String s) { System.out.println(NAME + ": " + s); }
}
```

Notes.

- A very clever use of recursion to skip unavailable destinations. The unavailable host causes the dispatch method called in go(URL) throw an exception. So ignore it and call go() again!
- clone() is used to copy the destination vector. Needed in connection with the recursion (to unwind the stack correctly).

Here is the travelling aglet itself. It doesn't actually do anything. If you run this example, look at the view->log of the Tahiti servers, and the messages in the corresponding command windows to see what is happening. Make sure to leave out at least one destination.

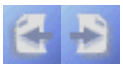
ItinerantAglet.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;
import java.util.*;

public class ItinerantAglet extends Aglet {

   Itinerary _itinerary = null;

   public void onCreation(Object init) {
      try {
         _itinerary = (Itinerary)init;
         addMobilityListener(
            new MobilityAdapter() {
               public void onArrival(MobilityEvent me) {
                  try {
                    if (_itinerary.hasMoreDestinations()) {
                        print("Going...");
                        _itinerary.go();
                    } else {
                        print("Done...");
                        dispose();
                    }
                  } catch (Exception e) {
                    print("Failed to dispatch.");
                    print(e.getMessage());
                  }
```

```
                }
             }
          );
          print("Going...");
          _itinerary.init(this);
       } catch (Exception e) {
          print("Failed to initalize the itinerary.");
          print(e.getMessage());
       }
    }
     static public String NAME = "ItinerantAglet";
    void print(String s) { System.out.println(NAME + ": " + s); }
}
```

Notes.

- Remember that parts of this code is executed on the home server, and parts at various servers on the trip. The parts in onArrival() are executed on remote sites in this case. (If the travelling aglet returned home, onArrival() would execute there too.) The line _Itinerary.init(this) is executed on the home server.
- The aglet code is very simple. To put itself in motion it just has to execute the lines highlighted by the different colour.
- To make the aglet do something, we need to combine the itinerary pattern with the master slave pattern. This is quite easy to do.



```
/*
 * @(#)Parent.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;
import java.util.*;

public class Parent extends Aglet {

   Itinerary _itinerary = null;

   public void onCreation(Object init) {
      try {
         Vector destinations = new Vector();
         destinations.addElement(new URL("atp://localhost:9000"));
         destinations.addElement(new URL("atp://localhost:9001"));
         destinations.addElement(new URL("atp://localhost:9002"));
         destinations.addElement(new URL("atp://localhost:9003"));
         destinations.addElement(new URL("atp://localhost:9004"));
         destinations.addElement(new URL("atp://localhost:9005"));
         URL origin = getAgletContext().getHostingURL();
         getAgletContext().createAglet(getCodeBase(),
                                       "aglets.agletbook.chapter8a.ItinerantAglet",
                                       new SeqItinerary(origin, destinations));
      } catch (Exception e) {
         print("Failed to initalize the itinerary.");
         print(e.getMessage());
      }
   }

   static public String NAME = "ParentItinerantAglet";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

```
/*
 * @(#)Itinerary.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import java.net.*;
import java.io.*;

public abstract class Itinerary implements Serializable {

   protected URL _origin = null;
   protected AgletProxy _aglet = null;

   public Itinerary(URL origin) {
      _origin = origin;
   }

   public void init(Aglet aglet) {
      _aglet = aglet.getAgletContext().getAgletProxy(aglet.getAgletID());
      go();
   }

   public URL getOrigin() {
      return _origin;
   }

   public AgletProxy getAglet() {
      return _aglet;
   }

   protected void go(URL destination) throws Exception {
      _aglet.dispatch(destination);
   }

   public abstract void go();

   public abstract boolean hasMoreDestinations();

   public abstract URL getNextDestination();
}
```

```
/*
 * @(#)SeqItinerary.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class SeqItinerary extends Itinerary {

   private Vector _destinations = null;

   SeqItinerary(URL origin, Vector destinations) {
      super(origin);
      _destinations = (Vector)destinations.clone();
   }

   public void go() {
      URL dest = getNextDestination();
      if (dest != null) {
         _destinations.removeElementAt(0);
         try {
            go(dest);
         } catch (Exception e) {
            print("Failed to dispatch (" + dest + ")");
            print("Skip this destination and go for the next.");
            print(e.getMessage());
            go();
         }
      }
   }

   public boolean hasMoreDestinations() {
      return _destinations.size() > 0;
   }

   public URL getNextDestination() {
      if (hasMoreDestinations())
         return (URL)_destinations.firstElement();
      else
         return null;
   }
```

```
   static public String NAME = "SeqItinerary";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

```
/*
 * @(#)ItinearntAglet.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;
import java.util.*;

public class ItinerantAglet extends Aglet {

   Itinerary _itinerary = null;

   public void onCreation(Object init) {
      try {
         _itinerary = (Itinerary)init;
         addMobilityListener(
            new MobilityAdapter() {
               public void onArrival(MobilityEvent me) {
                  try {
                    if (_itinerary.hasMoreDestinations()) {
                        print("Going...");
                        _itinerary.go();
                    } else {
                        print("Done...");
                        dispose();
                    }
                  } catch (Exception e) {
                    print("Failed to dispatch.");
                    print(e.getMessage());
                  }
               }
            }
         );
         print("Going...");
         _itinerary.init(this);
      } catch (Exception e) {
         print("Failed to initalize the itinerary.");
         print(e.getMessage());
      }
   }

    static public String NAME = "ItinerantAglet";
   void print(String s) { System.out.println(NAME + ": " + s); }
```

```
}
```

Master-Slave Itinerary Pattern

(Chapter 8)

This pattern creates a slave aglet which follows an itinerary, executing its task with doTask() on each server it comes to. The Slave2 class below, replaces the Slave1 class discussed previously.

Note that each time the slave does its task at a server, it immediately sends the result back a s a message to its master.

Slave2.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public abstract class Slave2 extends Aglet {

   Itinerary itinerary = null;
   AgletProxy parent = null;

   public void onCreation(Object args) {
      try {
         itinerary = (Itinerary)((Object[])args)[0];
         parent = (AgletProxy)((Object[])args)[1];

         initializeTask();

         addMobilityListener(
           new MobilityAdapter() {

             public void onArrival(MobilityEvent me) {
                print("Arrived...");
                try {
                  parent.sendMessage(new Message("Result",
doTask()));
                  if (itinerary.hasMoreDestinations()) {
                     print("Going...");
                     itinerary.go();
                  } else {
                     print("Done...");
                     dispose();
                  }
                } catch (Exception e) {
                    print("Failed to send result to master.");
```

```
                    print(e.getMessage());
                }
             }
          }
       );
       itinerary.init(this);
    } catch (Exception e) {
       print("Failed to create slave.");
       print(e.getMessage());
    }
 }
 abstract void initializeTask();
 abstract Object doTask();

 static public String NAME = "Slave2";
 void print(String s) { System.out.println(NAME + ": " + s); }
}
```

The Itinerary class and SeqItinerary class are unchanged..

A MyMaster class sets up the itinerary.

MyMaster1.java

```
package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;

public class MyMaster1 extends Aglet {
   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         Vector destinations = new Vector();
         destinations.addElement(new URL("atp://localhost:9000"));
         destinations.addElement(new URL("atp://localhost:9001"));
         destinations.addElement(new URL("atp://localhost:9002"));
         destinations.addElement(new URL("atp://localhost:9003"));
         destinations.addElement(new URL("atp://localhost:9004"));
         destinations.addElement(new URL("atp://localhost:9005"));
         URL origin = getAgletContext().getHostingURL();
         SeqItinerary itinerary = new SeqItinerary(origin,
destinations);
         AgletProxy thisProxy =
```

```
getAgletContext().getAgletProxy(getAgletID());
         Object[] args = new Object[] { itinerary, thisProxy };
         getAgletContext().createAglet(getCodeBase(),
                                       "aglets.agletbook.Slave2",
                                       args);
         print("Finished creating the child.");
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }
   public boolean handleMessage(Message msg) {
      if (msg.sameKind("Result")) {
         print("Received a result: \'" + msg.getArg() + "\'");
         return true;
      }
      return false;
   }
   static public String NAME = "MyMaster1";
   private void print(String s) { System.out.println(NAME + ": " +
s); }
   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch
(InterruptedException ie) { } }
}
```

```
<HTML>
<HEAD>
  <META NAME="GENERATOR" CONTENT="Adobe PageMill 3.0 Win">
  <TITLE>Aglet Patterns - Mater-Slave</TITLE>
                <link rel="stylesheet" href="cps.css">
        </HEAD>
<BODY BGCOLOR="#ffffff">

<H1><FONT COLOR="#7482cd">Master-Slave Pattern</FONT></H1>

<P>This is the simplest design pattern. Design patterns are discussed
under the following headings.</P>

<UL>
  <LI>Definition
  <LI>Purpose
  <LI>Applicability
  <LI>Participants
  <LI>Collaboration
  <LI>Consequences
  <LI>Implementation
</UL>

<H2>Definition</H2>

<P>A pattern where a master can delegate a task to a slave.</P>

<H2>Purpose</H2>

<P>A task is split between two computers. A parallel process is
possible. While the slave is away doing its task, the master can
continue with its task. The slave usually sends the result of
its taks back to the master.</P>

<H2>Applicability</H2>

<P>parallleism is needed</P>

<P>the master needs to get something done on another machine</P>

<H2>Participants</H2>

<UL>
  <LI>Abstract slave
  <LI>Concrete slave
  <LI>Master
</UL>

<H2>Collaboration</H2>

<P>The master-slave participants cooperate as follows:</P>

<OL>
  <LI>A master aglet creates a slave aglet
  <LI>The slave initializes its task
  <LI>The slave moves to a remote host and executes its task.
  <LI>The slave sends the result of its task to the master
  <LI>The slave disposes itself
</OL>

<H2>Consequences</H2>
```

```
<P>The constant parts of the design are separated from the variable
parts. The constant parts need only be implemented once, and developers
can concentrate on the variable point.</P>

<H2>Implementation</H2>

<P>(Aglet book, chapter 8)</P>

<P>The implementation is built aound a foundation abstract class:
(called Slave.java in the textbook).<A NAME="anchor76871"></A></P>

<P><A HREF="sourceCode/patterns/masterSlave/Slave1.java">Slave1.java</A></P>

<PRE><B><CODE><FONT COLOR="#ff0000">package aglets.agletbook.chapter8;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import com.ibm.aglet.*;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import com.ibm.aglet.event.*;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import java.net.*;</FONT></CODE></B>

<B><CODE><FONT COLOR="#ff0000">public </FONT><FONT COLOR="#990000">abstract</FONT><FONT
COLOR="#ff0000"> class Slave1 extends Aglet {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   URL destination = null;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   AgletProxy master = null;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   public void onCreation(Object args) {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      try {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         destination =
(URL)((Object[])args)[0];</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         master =
(AgletProxy)((Object[])args)[1];</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         </FONT><FONT
COLOR="#990000">initializeTask();</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         addMobilityListener(</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">            new MobilityAdapter() {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">               public void onArrival(MobilityEvent me)
{</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">
print(&amp;quot;Arrived...&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                  try {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                     master.sendMessage(new
Message(&amp;quot;Result&amp;quot;, </FONT><FONT COLOR="#990000">doTask()</FONT><FONT
COLOR="#ff0000">));</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                     dispose();</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                  } catch (Exception e) {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                     print(&amp;quot;Failed to send result to
master.&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">
print(e.getMessage());</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">                  }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">               }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">            }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         );</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         dispatch(destination);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      } catch (Exception e) {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(&amp;quot;Failed to create
slave.&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(e.getMessage());</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   }</FONT></CODE></B>
<B><CODE><FONT COLOR="#990000">   abstract void initializeTask();</FONT></CODE></B>
<B><CODE><FONT COLOR="#990000">   abstract Object doTask();</FONT></CODE></B>
```

```
<B><CODE><FONT COLOR="#ff0000">   static public String NAME =
&amp;quot;Slave1&amp;quot;;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   void print(String s) { System.out.println(NAME +
&amp;quot;: &amp;quot; + s); }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">}</FONT></CODE></B></PRE>

<P>The slave's mobility action on arrival is always the same,
so we abstract it out in this abstract class. The same is true
of the slave's messaging action.</P>

<P>What is different for each slave is (1) how it is initialzed,
and (2) what its task is. The corresponding methods initializeTask()
and doTask() are implemented in a subclass of this one.</P>

<P>Lange and Oshima provide this simple example. The abstraction
is made concrete.</P>

<P>In this example, the task is just to print a message to stdout
on the receiver, and send a string back to the master aglet at
home.</P>

<P><A HREF="sourceCode/patterns/masterSlave/MySlave.java">MySlave.java</A></P>

<PRE><B><CODE><FONT COLOR="#ff0000">package
aglets.agletbook.chapter8.MySlave;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import com.ibm.aglet.*;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import com.ibm.aglet.event.*;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import java.net.*;</FONT></CODE></B>

<B><CODE><FONT COLOR="#ff0000">public class MySlave </FONT><FONT COLOR="#0000ff">extends
Slave1</FONT><FONT COLOR="#ff0000"> {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   public void </FONT><FONT
COLOR="#990000">initializeTask()</FONT><FONT COLOR="#ff0000"> {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      print(&amp;quot;Initializing.&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   public Object </FONT><FONT
COLOR="#990000">doTask()</FONT><FONT COLOR="#ff0000"> {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      print(&amp;quot;Performs task&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      return &amp;quot;Some result...&amp;quot;;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   static public String NAME =
&amp;quot;MySlave&amp;quot;;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   void print(String s) { System.out.println(NAME +
&amp;quot;: &amp;quot; + s); }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">}</FONT></CODE></B></PRE>

<P>A master aglet at home creates the concrete slave and sends
it off to do its thing.</P>

<P><A HREF="sourceCode/patterns/masterSlave/MyMaster.java">MyMaster.java</A></P>

<PRE><B><CODE><FONT COLOR="#ff0000">package aglets.agletbook.chapter8;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import com.ibm.aglet.*;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import java.net.URL;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">import java.util.*;</FONT></CODE></B>

<B><CODE><FONT COLOR="#ff0000">public class MyMaster extends Aglet {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   public void run() {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      print(&amp;quot;STARTING
--------------------&amp;quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      try {</FONT></CODE></B>
```

```
<B><CODE><FONT COLOR="#ff0000">         print(&quot;Creating the
child...&quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         String host =
getAgletContext().getHostingURL().toString();</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         URL destination = new
URL(&quot;atp://proton.scs.ryerson.ca:9000&quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         AgletProxy thisProxy =
getAgletContext().getAgletProxy(getAgletID());</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">//         SlaveSetup setup = new SlaveSetup(destination,
thisProxy);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         Object[] args = new Object[] { destination,
thisProxy };</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">
getAgletContext().createAglet(getCodeBase(),</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">
&quot;aglets.agletbook.chapter8.MySlave&quot;,</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">
args);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(&quot;Finished creating the
child.&quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      } catch (Exception e) {</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(&quot;Failed to create the
child.&quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(e.getMessage());</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   public boolean handleMessage(Message msg)
{</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      if
(msg.sameKind(&quot;Result&quot;))</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">         print(&quot;Received a result: \'&quot; +
msg.getArg() + &quot;\'&quot;);</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">      return true;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   static public String NAME =
&quot;MyMaster&quot;;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   private void print(String s) { System.out.println(NAME
+ &quot;: &quot; + s); }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   private static long SLEEP = 3000;</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">   private void pause() { try { Thread.sleep(SLEEP); }
catch (InterruptedException ie) { } }</FONT></CODE></B>
<B><CODE><FONT COLOR="#ff0000">}</FONT></CODE></B></PRE>

<P>A typical ouput on the home server would be:</P>

<DL>
  <DD>MYMASTER: STARTING ----------------------
  <DD>MYMASTER: Creating the child ...
  <DD>MYMASTER: Finished createing the child.
  <DD>MySlave: Intitializing ...
  <DD>MyMaster: Received a result: 'Some result ...'
</DL>

<P> </P>

<P> </P>

<P> </P>

<P> 
```

```
</BODY>
</HTML>
```

```
/*
 * @(#)Slave2.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package aglets.agletbook.chapter8a;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public abstract class Slave2 extends Aglet {

   Itinerary itinerary = null;
   AgletProxy parent = null;

   public void onCreation(Object args) {
      try {
         itinerary = (Itinerary)((Object[])args)[0];
         parent = (AgletProxy)((Object[])args)[1];
         initializeTask();
         addMobilityListener(
           new MobilityAdapter() {
             public void onArrival(MobilityEvent me) {
                print("Arrived...");
                try {
                  parent.sendMessage(new Message("Result", doTask()));
                  if (itinerary.hasMoreDestinations()) {
                     print("Going...");
                     itinerary.go();
                  } else {
                     print("Done...");
                     dispose();
                  }
                } catch (Exception e) {
                    print("Failed to send result to master.");
                    print(e.getMessage());
                }
              }
            }
         );
         itinerary.init(this);
      } catch (Exception e) {
         print("Failed to create slave.");
         print(e.getMessage());
      }
   }
```

```
   abstract void initializeTask();

   abstract Object doTask();

   static public String NAME = "Slave2";
   void print(String s) { System.out.println(NAME + ": " + s); }
}
```

```
/*
 * @(#)MyMaster1.java
 *
 * (c) Copyright Danny B. Lange & Mitsuru Oshima, 1998
 *
 * THIS ROGRAM IS PROVIDED "AS IS" WITHOUT ANY WARRANTY
 * EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
 * THE WARRANTY OF NON-INFRINGEMENT AND THE WARRANTIES
 * OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 * THE AUTHORS WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED
 * BY YOU AS A RESULT OF USING THIS SAMPLE PROGRAM. IN NO
 * EVENT WILL THE AUTHORS BE LIABLE FOR ANY SPECIAL,
 * INDIRECT CONSEQUENTIAL DAMAGES OR LOST PROFITS EVEN IF
 * THE AUTHORS HAS BEEN ADVISED OF THE POSSIBILITY OF
 * THEIR OCCURRENCE OR LOSS OF OR DAMAGE TO YOUR RECORDS
 * OR DATA. THE AUTHORS WILL NOT BE LIABLE FOR ANY THIRD
 * PARTY CLAIMS AGAINST YOU.
 */

package agletbook;

import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;

public class MyMaster1 extends Aglet {

   public void run() {
      print("STARTING --------------------");
      try {
         print("Creating the child...");
         Vector destinations = new Vector();
         destinations.addElement(new URL("atp://localhost:9000"));
         destinations.addElement(new URL("atp://localhost:9001"));
         destinations.addElement(new URL("atp://localhost:9002"));
         destinations.addElement(new URL("atp://localhost:9003"));
         destinations.addElement(new URL("atp://localhost:9004"));
         destinations.addElement(new URL("atp://localhost:9005"));
         URL origin = getAgletContext().getHostingURL();
         SeqItinerary itinerary = new SeqItinerary(origin, destinations);
         AgletProxy thisProxy = getAgletContext().getAgletProxy(getAgletID());
         Object[] args = new Object[] { itinerary, thisProxy };
         getAgletContext().createAglet(getCodeBase(),
                                       "aglets.agletbook.Slave2",
                                       args);
         print("Finished creating the child.");
      } catch (Exception e) {
         print("Failed to create the child.");
         print(e.getMessage());
      }
   }

   public boolean handleMessage(Message msg) {
      if (msg.sameKind("Result")) {
         print("Received a result: \'" + msg.getArg() + "\'");
         return true;
      }
      return false;
   }

   static public String NAME = "MyMaster1";
```

```
   private void print(String s) { System.out.println(NAME + ": " + s); }
   private static long SLEEP = 3000;
   private void pause() { try { Thread.sleep(SLEEP); } catch (InterruptedException ie) {
} }
}
```

Aglet API Sequential Itinerary

The Aglet API provides implimentations of a number of aglet patterns. These are in the packages com.ibm.agletx.util and com.ibm.agletx.pattern.

Here is an example using the SeqItinerary class and its child class, SlaveItinerary. An aglet is sent around an itinerary. It sends a message back to its master when it reaches its final destination.

ItinerantAglet3.java

```
package aglets.mystuff.chapter8c;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import com.ibm.agletx.util.*;

public class ItinerantAglet3 extends Aglet {

   SlaveItinerary slaveTrip = null;
   AgletProxy parent = null;

   public void onCreation(Object ini) {

      parent = (AgletProxy) ini;
      slaveTrip = new SlaveItinerary(this, "", new MyTask());

      slaveTrip.addPlan("atp://localhost:9000");
      slaveTrip.addPlan("atp://localhost:9001");
      slaveTrip.addPlan("atp://localhost:9002");
      slaveTrip.addPlan("atp://localhost:9003");
      slaveTrip.addPlan("atp://localhost:9004");

      slaveTrip.startTrip();
   }
   public void run() {
     .
      if(slaveTrip.atLastDestination()) {
         try {
            parent.sendOnewayMessage(new Message("finished", "At last stop"));
         } catch(Exception e) {
            System.out.println("Message failed");
         }
         finally {
            dispose();
         }
      }
   }

   class MyTask extends Task {
      public void execute(SeqItinerary i) {
         System.out.println("George was here");
```

```
        }
    }
}
```

- The SeqItinerary and SlaveItinerary classes provide a number of useful methods. Note also the Task class which has only one method, void execute(SeqItinerary). As you can see, you don't actually call this yourself.
- The SlaveItineray class is smart enough to skip unavailable destinations automatically.
- Another useful class in the agletx.util package is the SeqPlanItinerary class. See the documentation.

Here is a parent class to send the above aglet on its way.

Parent3.java

```
package aglets.mystuff.chapter8c;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import com.ibm.agletx.util.*;

public class Parent3 extends Aglet {

        private AgletProxy slaveAgletProxy = null;

        public void onCreation(Object init) {
                try {
                        slaveAgletProxy = getAgletContext().createAglet(

getCodeBase(),

"aglets.mystuff.chapter8c.ItinerantAglet3",

this.getProxy());
                } catch (Exception e) {}

                addMobilityListener(new StopDispatch());
        }

        public boolean handleMessage(Message msg) {
                if(msg.sameKind("finished")) {
                        System.out.println("George says: " + msg.getArg());
                        return true;
                }
                return false;
        }

        class StopDispatch extends MobilityAdapter {
                public void onDispatching(MobilityEvent me) {
                        setText("Sorry, I'm immobile. You killed me!");
                        try {
                                Thread.sleep(2000);
                        } catch(InterruptedException e) {}
                        dispose();
```

```
                }
        }
}
```

This class also shows a simple way to keep an aglet immobile.

```
/**
 * ItinerantAglet.java
 * Uses the agletx.util package classes, SeqItinerary, SlaveItinerary, and Task.
 * Compare this to the methods in Chapter 8 of Lange and Oshima.
 * This aglet is created by Parent3 aglet in Parent3.java.
 * ItinerantAglet goes to the hosts listed below, prints a message on each. At the
 * last host in the trip it also sends a string message back home.
 * DG. Oct. 99
 */


package aglets.mystuff.chapter8c;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import com.ibm.agletx.util.*;

public class ItinerantAglet3 extends Aglet {

   SlaveItinerary slaveTrip = null;
   AgletProxy parent = null;

   public void onCreation(Object ini) {

      parent = (AgletProxy) ini;

      slaveTrip = new SlaveItinerary(this, "", new MyTask());
      slaveTrip.addPlan("atp://localhost:9000");
      slaveTrip.addPlan("atp://localhost:9001");
      slaveTrip.addPlan("atp://localhost:9002");
      slaveTrip.addPlan("atp://localhost:9003");
      slaveTrip.addPlan("atp://localhost:9004");

      slaveTrip.startTrip();

   }

   public void run() {
      // Do not put the following in the MyTask class.
      if(slaveTrip.atLastDestination()) {
         try {
            parent.sendOnewayMessage(new Message("finished", "At last stop"));
         } catch(Exception e) {
            System.out.println("Message failed");
         }
         finally {
            dispose();
         }
      }
   }

   class MyTask extends Task {
      public void execute(SeqItinerary i) {
         System.out.println("George was here");
      }
   }

}
```

```
package aglets.mystuff.chapter8c;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import com.ibm.agletx.util.*;

public class Parent3 extends Aglet {


   private AgletProxy slaveAgletProxy = null;

   public void onCreation(Object init) {

      try {
         slaveAgletProxy = getAgletContext().createAglet(
                           getCodeBase(),
                           "aglets.mystuff.chapter8c.ItinerantAglet3",
                           this.getProxy());
      } catch (Exception e) {}

      addMobilityListener(new StopDispatch());
   }

   public boolean handleMessage(Message msg) {
      if(msg.sameKind("finished")) {
         System.out.println("George says: " + msg.getArg());
         return true;
      }
      return false;
   }

   class StopDispatch extends MobilityAdapter {
      public void onDispatching(MobilityEvent me) {
         setText("Sorry, I'm immobile. You killed me!");
         try {
            Thread.sleep(2000);
         } catch(InterruptedException e) {}
         dispose();
      }
   }
}
```

Aglets as Agents

Earlier in these notes we discussed Russell and Norvig's agent classification scheme and looked at Nilsson's reactive wall following robot agent. How do aglets fit into these AI pictures?

Aglets live in an environment provided by the Tahiti servers. Aglets have sensors and effectors, which are various methods provided by the aglet API. Among these are:

Sensors

- Object getProperty(String). A method of the AgletContext class which the aglet can access by calling getContext()
- Various methods of the Reader and InputStream classes and their subclasses, with the permission of the Tahiti server security manager.
- boolean handleMessage(Message). A method of the Aglet class which allows an aglet to detect messages sent by other aglets.

Effectors

- void setProperty(String, Object). This method allows the aglet to change its immediate environment.
- Various methods of the Writer and OutputStream classes and their subclasses, with the permission of the Tahiti server security manager.
- Object sendMessage(Message). Allows the aglet to effect the behavioiur of other aglets. There aer several variations on this method supplied by the API.

The basic aglet is a reactive agent with state. It has the added feature of mobility, and thanks to the Internet, the ability to influence remote environments as well as local ones.

KQML in InfoSleuth

Here is an excerpt from the paper, Nigel Jacobs and Ray Shea, The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources. Click here for the whole original paper. InfoSleuth is an experimental multi-agent system. The system is shown in Figure 3 below.

5 Query Views

Query views are database applications implemented as Java applets which can be used to retrieve and manipulate data from the network. Query views take advantage of the core capabilities that Java has to offer.

Query view applets are written on top of a query API layer which provides an abstract, high-level method for modifying ontologies, constructing queries against the ontologies, executing the queries, and retrieving the results. The underlying functionality of the API is actually carried out by the agent network; in essence, the API is a wrapper to the query implementation provided by the network of query agents.

Query views can implement either general purpose or domain-specific metaphors for query construction and data visualization. The viewer server, which maintains the repository of query views, can fulfill requests for query view applets based on the desired functionality and the ontology or domain model which the query view supports. When the user selects a domain (or ontology) and a task or set of tasks he wishes to accomplish, the user agent retrieves the necessary set of applets for accomplishing this from one or more viewer servers on the network, and allow the user to load and use these applets, then discard them.

This is a very powerful paradigm. With Java, we have extended the notion of using a network of agents to find and retrieve data, and now use that same network of agents to dynamically find, retrieve, and load the proper GUI applets for interacting with that data, based on the task domain and qualities of the data itself. In a sense, we are automating the software distribution process using the same techniques with which we are automating the data retrieval process.

6 Physical Agent Architecture

One can think of the InfoSleuth network as a cloud of agents (Figure 3), through which data passes back and forth between users and data resources. All communication within this cloud is conducted by means of KQML and high-level ontologies. The user interacts with data resources (the existence of which may be unknown at the time requests are made) by passing requests into the cloud via his personal user agent, with which he communicates via Java applets. At the other end, a data resource (for example, an Oracle database) accepts requests from the cloud via its own resource agent, which translates the KQML/ontology-based query into the query language understood by the local resource (for example, SQL).

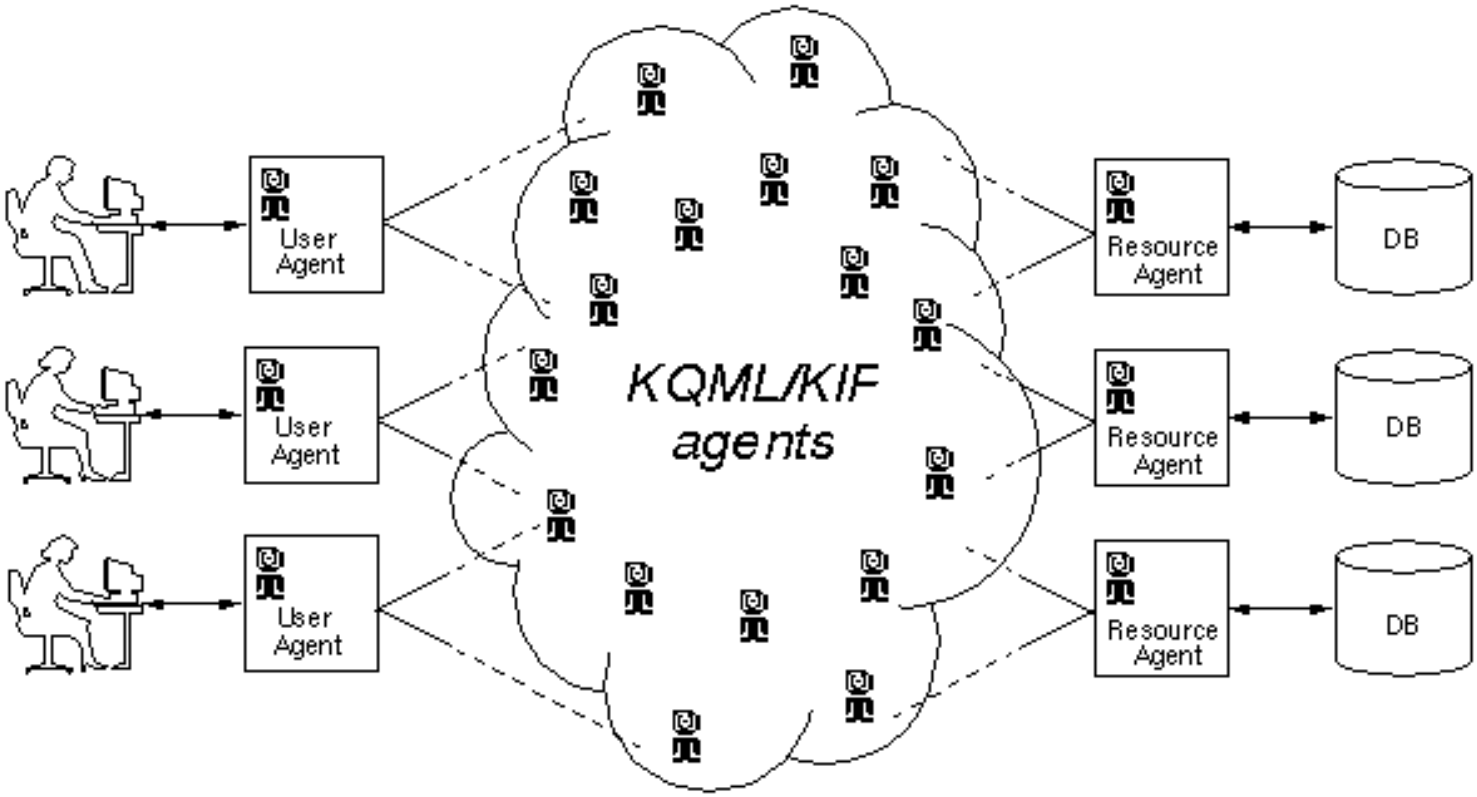


Figure 3. Cloud of Agents

Figure 4 shows a simple implementation of this network of agents, with the cloud of agents actually providing only a single thin layer between the user agents and resource agents. Since the communication mechanism between all agents is based on KQML, and since an agent can participate simply by advertising its services to a broker, it is a simple matter to integrate other KQML-aware agents into the system, thus providing a high degree of extensibility.

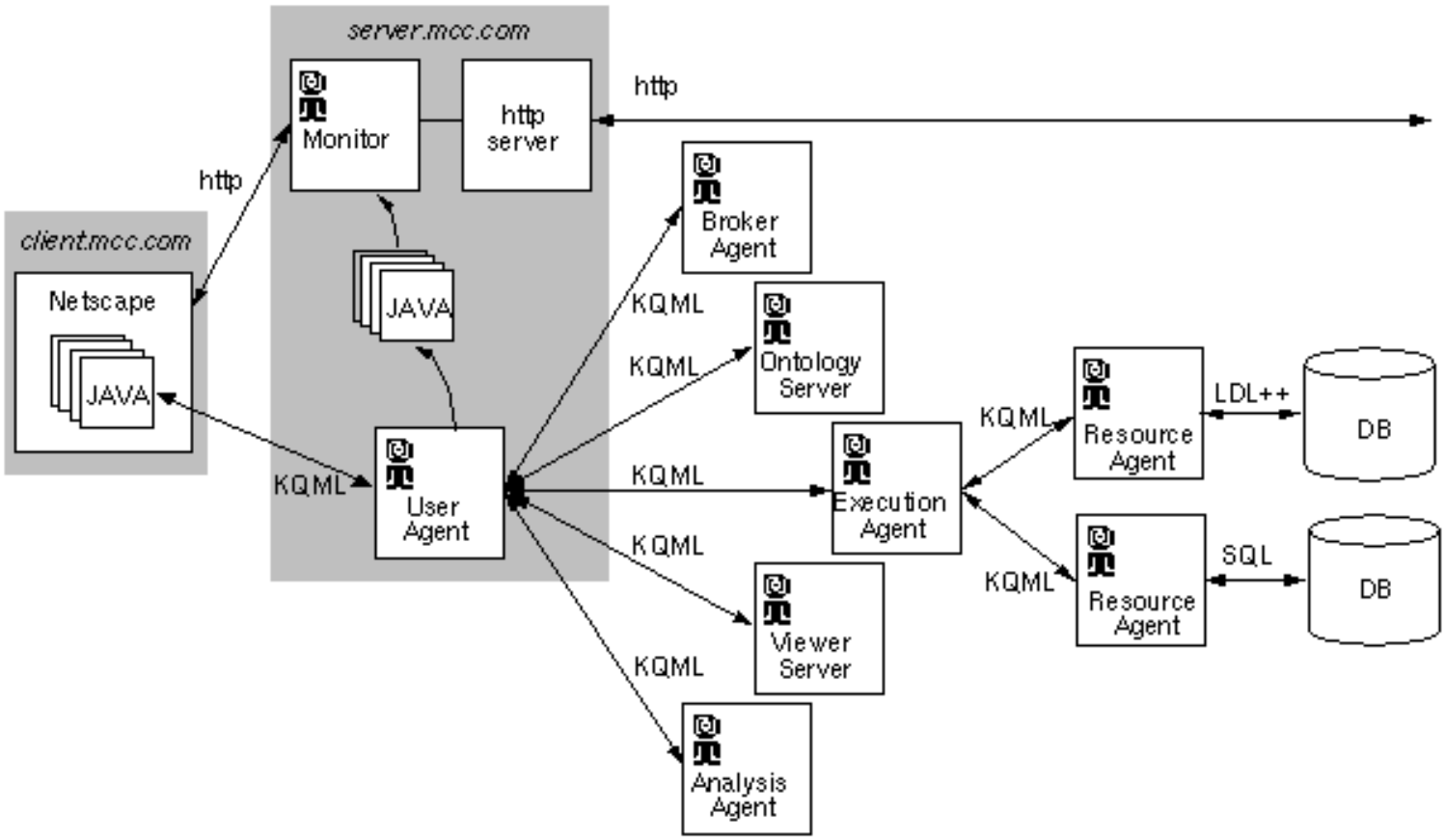


Figure 4. A Simple Agent Network

Each of the agents depicted in Figure 4 is capable of handling multiple user sessions, except for the user agent, which is intended to serve as a personal agent to a single user (although it can manage multiple sessions with other agents).

Following is an overview of the function of each agent.

6.1 User Agent

The user agent is the user's intelligent gateway to the network of InfoSleuth agents. As such, it is primarily responsible for handling requests from the user via Java applets, routing those requests to appropriate server agents, and passing responses back to the user. The user agent is persistent and autonomous, thus it is able to maintain the user's context beyond a single browser session, allowing long-term autonomous data retrieval and other tasks to continue in the user's absence. It is capable of storing information (data and queries) for the user, maintaining a user model, and can act as a resource for other agents, for instance as a means of sharing information with other user agents. The user agent is implemented as a stand-alone Java application.

6.2 Monitor

The monitor is an HTTP proxy which serves two roles:

- · monitor user web access and report to the user agent for later inferencing and pattern detection.
- · accept Java applets via the user agent and place them in the proper directories so that they can be accessed by the user. (See Section 9, "Security Concerns").

This agent is closely tied to the user agent, and like it, is implemented in Java.

6.3 Broker Agent

The broker agent acts as a matchmaker which pairs requests from agents to other agent services that can fulfill that request. As agents come on line, they can advertise their services to the broker via KQML. Any agent can ask the broker for a recommendation on how to fulfill a particular request, and the broker will respond with the addresses of the agents that have advertised that service. Possible future capabilities for the broker include delegation (i.e., "passing the buck"), and subscription, allowing requesting agents to subscribe to various kinds of information, enabling asynchronous notification when the desired resources become available. The broker is a Java application.

6.4 Ontology Server

The ontology server is responsible for managing the creation, update and querying of multiple ontologies. KIF is used both to query the ontologies and to express the query results. Ontologies may be imported and exported in KIF and several other representation languages. Different ontology formats (e.g., relational or object database schema, entity-relationship models) are described via ontology meta-models. Ontologies may be nested, and references may be made between ontologies. The ontology server maintains internal consistency. Many ontology servers may be deployed, with each server advertising the ontologies it maintains via the broker agents. The server is currently implemented as a Java application.

6.5 Execution Agent

The execution agent is responsible for high-level execution of ontology-based queries. It accepts KQML messages containing KIF-based queries, decomposes the queries into sub-queries based on its knowledge of appropriate resource agents that can satisfy the query, and sends the high-level sub-queries off to the resource agents. It then can merge the results received and transmit them to the agent which originated the query. The execution agent is implemented in Java with embedded CLIPS functions.

6.6 Resource Agents

The resource agent is to the local database what the user agent is to the user. It acts as an intelligent front-end interface for the relatively dumb DBMS or other data store, accepting high-level ontology-based queries from the network and translating them into whatever local query language (e.g., SQL) is necessary to execute the queries against the local database. Results are then translated back into the terms of the ontology and returned to the requesting agent. Just as the user agent maintains the user context, the resource agent is able to maintain resource context, allowing for incremental retrieval of query results by requesting agents. Additionally, resource agents are able to obtain, store, and advertise meta-information about the contents of their local resource. Current resource agents are implemented in Java and LDL++, with ODBC and SQL versions under development, and JDBC versions [13] planned.

6.7 Data Analysis Agents

Data analysis agents perform various analysis, knowledge mining, and pattern recognition tasks on data returned by a query. These agents are implemented in Java, CLIPS, LDL++, and LISP.

6.8 Viewer Server

The viewer server is a specialized resource agent, in that it maintains a storehouse of Java applets which are designed to manipulate KIF ontologies and queries based on those ontologies, via a standard InfoSleuth

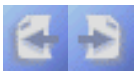
meta-model query API. In a sense, the viewer server is an applet broker which serves database applications, query editors, and visualization tools for manipulating and displaying query results. It is implemented as a stand-alone Java process.

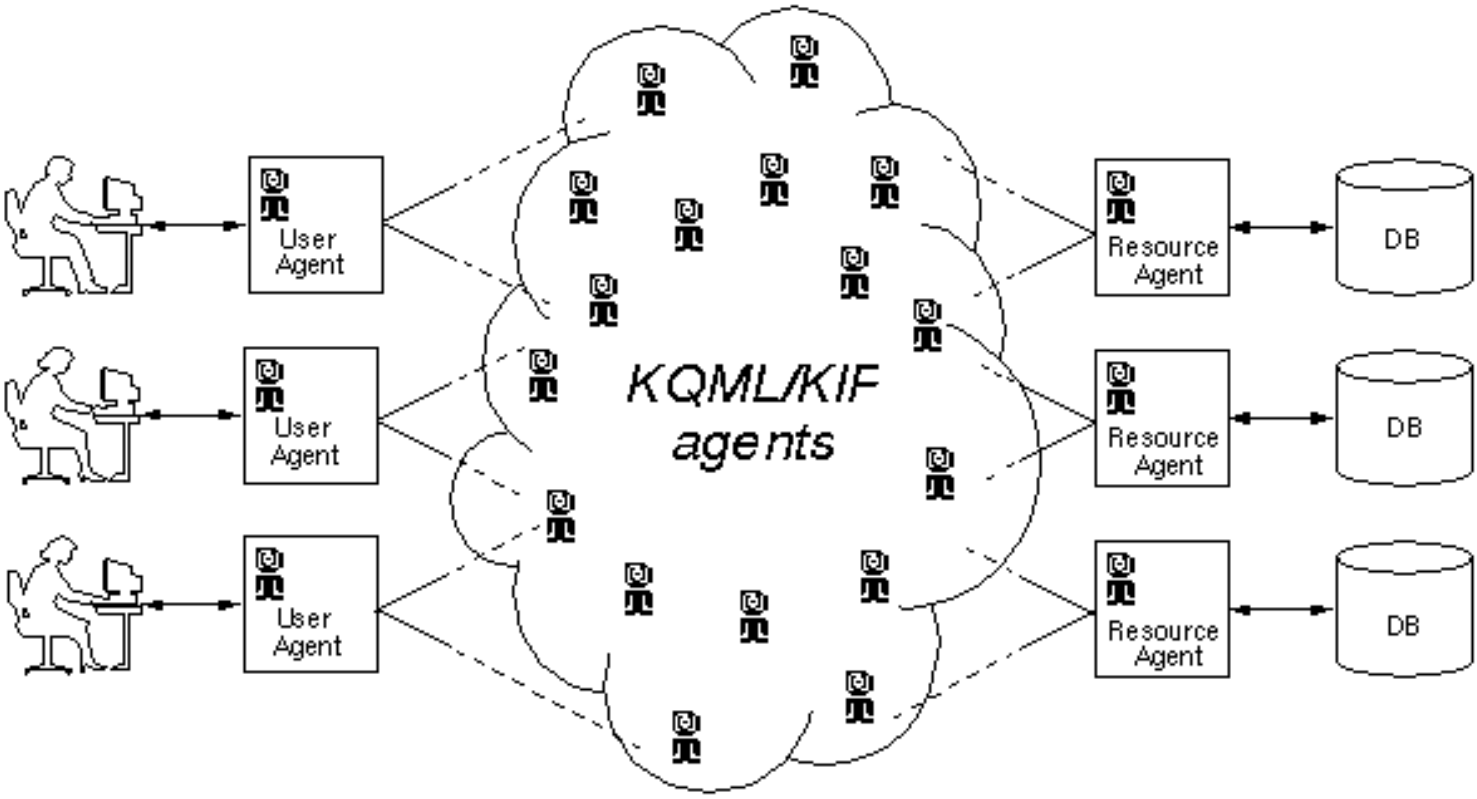
7 Sample Scenario

Suppose the user is interested in locating real estate data stored in a variety of distributed databases, based on various factors such as price, location, zoning, etc. Using InfoSleuth, she logs in to her user agent via a login applet in Netscape, and requests recommendations for ontologies and applications that are appropriate to the problem domain. The user agent queries a broker agent for servers that could fulfill the request for ontologies, and is directed to an ontology server. The ontology server recommends a list of ontologies it knows about that relate to the problem domain, and the user agent allows the user to select one.

Now that the particular ontology is selected, the user agent again asks the broker to recommend a server that can supply query and visualization applets that work with the selected ontology. Again the broker recommends a viewer server, the user agent sends the request to the viewer server, and the server responds by sending one or more Java applets that fulfill the requirements. Assuming verification of the trustworthiness of the applets is done (see Section 9, "Security Concerns"), the user agent passes the applets to the security monitor, which places the applets in the appropriate codebase hierarchy for access by the user. At this point the user agent send the names of the applets to the login applet, which permits the user to load them.

The user iteratively constructs an appropriate query via the domain-specific applet, using concepts from the selected ontology. She can then submit the queries for execution. The user agent consults the broker for recommendations on how to execute the query, and is directed toward an execution agent. The execution agent decomposes the query into sub-queries, submits the sub-queries to one or more resource agents, which resolve the sub-queries against their respective local databases and return the results to the execution agent, which collates the results and passes them back, via the user agent, to the user.




User Agent
User Agent
User Agent
KQML/KIF agents
Resource Agent
Resource Agent
Resource Agent
DB
DB
DB

The Web's Next Incarnation: Intelligent Talk

By Tim McDonald
NewsFactor Network
November 13, 2001

Send this Article | Talkback
Print this Article | Related Stories

Teaching logic to machines and systems while maintaining flexibility is a tall order, and critics of the Semantic Web say it cannot be done.

The latest effort to organize the Web's vast store of information is called the "Semantic Web," and while it remains to be seen whether it can live up to its billing, it is promising enough to have attracted scientists from a variety of disciplines, including Tim Berners-Lee, director of the World Wide Web Consortium (W3C) at the Massachusetts Institute of Technology.

The Semantic Web hopes to make our Web experience better by enabling our machines to talk intelligently with other machines. It would be an extension of the current Web, a place where "information is given well-defined meaning, better enabling computers and people to work in cooperation," according to Berners-Lee.

"The Semantic Web is really data that is processable by machine," Berners-Lee says. "That's what the fuss is about."

'Talk' Instead of 'Link'

Today's Web is basically a "publishing medium," a huge warehouse where text and images are stored. The Semantic Web wants to turn it into a more interactive place, where information can he interpreted and exchanged and where software agents roam from page to page, performing sophisticated

Alt Text

November 17, 2001

NAS	1898.58	-1.99
S&P	1138.65	-3.59
DOW	9866.99	-5.40

'Harry Potter' Movie Ticket Sales Break Internet Records
Full Story

U.S. Defends Microsoft Settlement, Rejects Penalties
Full Story

Senate Approves Internet Tax Ban Extension
Full Story

Renewed Investor Optimism About Tech Stocks
Full Story

European Union Eyes Cookie Limits
Full Story

Exclusive NewsFactor Interview with Intel President & CEO Craig Barrett (Part 2)
Full Story

High-Speed Mini-Stereo Big on Web Entertainment
Full Story

CRM: Efficiency Is Not

tasks for users.

Instead of merely displaying information on their screens, computers will "understand" what they are displaying.

"Ultimately, we'll be able to utilize a series of helpers to help us manage our day-to-day activities and automate a lot of the things we do -- calendaring, coordination, resource discovery -- things like that," Eric Miller, head of the W3C Semantic Web's effort at MIT, told NewsFactor Network.

"Right now, the Web works by allowing people all over the world to link to each other," Miller said. "Current technology allows us to say "links to," and what we really want to say is "talks to."

"It's a way of providing some additional contextual relationships with the things we're interacting with daily. It helps make it clear to machines and humans how these things relate."

Commercials to Keats

In order for the Semantic Web to work, computers will need a common vocabulary as well as rules. The Web now contains mostly documents written for people, rather than data and information that can be processed automatically by the Semantic Web.

Computers simply cannot comprehend rich, varied and often confusing human language, which ranges from the mundane -- tire commercial text -- to the lofty -- the poetry of John Keats.

Computer language that will help the Semantic Web evolve already exists, experts say, in the form of Extensible Markup Language, which gives more structure to Web pages.

Resource Description Framework (RDF) is the language of the Semantic Web in much the same way that HTML is the language of the current Web. RDF integrates information from multiple sources, and is itself a framework for "metadata" -- data about data.

Logical and Flexible

According to the W3C, computers must have access to "structured collections of information and sets of inference rules that they can use to conduct automated reasoning."

Artificial intelligence experts have studied this field for decades. Such systems are often called "knowledge representation," and have traditionally been very centralized -- where everyone shares exactly the same definition of specific words, like "head" or "director."

These systems limit what questions can be asked so that the computer can answer correctly, if it answers at all.

Teaching logic to machines and systems while maintaining flexibility is a tall order, and critics of the Semantic Web say it cannot be done. But Miller says that by taking it slowly, it can.

"We have very much in the Semantic Web an eye toward the future goals through well-established incremental steps," Miller said. "We have the notion of making the simple steps simple, and the complex stuff possible." END

Talkback: Click here to add your comment about this story...

See Related Stories

Working
Full Story

Music Retailer Embarks on Customer Loyalty Offensive
Full Story

Satellite Service Delivers Broadband Wireless to the Fast Lane
Full Story

Gateway Makes House Calls for Wireless Networking
Full Story

Dell Defies Poor Economy To Chart Profits, Market Gains
Full Story

Whatever Happened to Dot-Com Stunts?
Full Story

Tech Innovators Learn How To Avoid Washing Out
Full Story

Motorola To Offer New Mobile Messaging System
Full Story

Starwood Cultivates Loyalty with New Web Site
Full Story

Can We Stop the Terrorist Tech Trade?
Full Story

Gates Sells First Xbox Game Console
Full Story

Hewlett-Packard Earnings May Doom Union with Compaq
Full Story

Yahoo! To Cut Workforce 10 Percent
Full Story

Amazon Reorganizes, Emphasizing Third-Party Services
Full Story

FBI: Old-Tech Fingerprints Still Best Clues
Full Story

After the Fall: The Future of CRM, Part 4
Full Story

Headset Highlighted By What You Don't Hear
Full Story

Shipping Just Gets Harder for E-tailers
Full Story

Tech Cartoon
Just for Fun

Friday's Cybercrime Report
Full Story

Personalized Web Site Features - Customer Picks and Pans
(12-Nov-01)
Web Trackers: The Spies in Your Computer
(08-Nov-01)
Internet Heavyweights Seek Profit in Security
(05-Nov-01)
The Internet Is an Open Book - Protect Yourself with Secure Protocols
(02-Nov-01)

See more news
Get news by e-mail
Visit open forums

Communicative Acts

In the Aglet examples, communcation among agents is purely syntactic. It consists of matching strings. These strings carry no meaing as far as the Aglets are concerned. If artificial agents are to have more sophisticated communication among themselves, communication which carries meaning, more complex communication methods are needed.

A Communicative Act (a "language game")/

As usual in AI, researchres look to human examples. In the case of communication they found a suitable theory in so-called speech acts, a development of linguistic philosophy.

In these notes we first look at this philosophical background. Then we see how FIPA, the Foundation for Intelligent Physicall Agents has adapted speech acts to the computer world to create an international standard agent commuication language (ACL). We also look at a related standard, the Semantic Language (SL) used to describe the content of communications. Finally, we look at an actual system, JADE, the Java Agent Development Envirionment, which implements these ideas.

Some Philosophical Background

John Searle's Contribution

More on Speech (Communication) Acts.

Summary

Speech Acts, background

John Searle developed his speech act theory in the late 1960's. He derived it from ideas of his teacher, John L, Austin, who in turn was influenced by the anglo-austrian philosopher, Ludwig Wittgenstein. These philosophers developed their theories in part in opposition to another philosophical school, the Logical Positivists. Logical Positivism was developed by a group of philosophers called the Vienna Circle in the 1920's and 1930's.

The Question of Meaning

Logical Positivism

This is a very uncompromising philosophy. It is based on the ***principle of verification***. There are only ***two sources of real knowledge: (1) logic (2) empirical observation***. Anything else is meaningless conjecture.

Logic follows strict rules of proof and tests of internal consistency. For example, the statement "I will meet you yesterday." is nonsensical and meaningless. We know this without reference to anything other than the meanings of the words. It is illogical, internally inconsistent. The Logical Positivists would say we are using analytical knowledge when we analyze the possible meanings of this sentence.

An example of empirically observed knowledge is: At the surface of the earth, the acceleration due to gravity is 9.8 ms-2. This claim can be verified by experiment. (Perhaps you did just this in high school.) This category of knowledge, the logical positivists referred to as synthetic knowledge.

As you can see, the logical positivists accepted only scientific and mathematical knowledge as valid. In their view, any statement which could not be tested either via logic, or via experiment, was meaningless. They called such statements metaphysical. An example of a metaphysical (and therefore meaningless in their view) is "God exists". Another example: "God does not exist".

Scientific and Everyday Language

What the logical positivists were doing was ***privileging scientific language*** (logic, mathematics) above ordinary natural languages (english, mandarin). Scientific statements were either true, or false, and could be verified to be one or the other. Natural language was ambiguous and often just a babble of unfounded opinions as far as the logical positivists were concerned. Ordinary language was useful for ordinary life but not for serious thought.

If this philosophy sounds extreme, it is. But many people with science and engineering backgrounds often hold this philosophy although they probably won't admit it. It comes out when the so-called "hard" sciences sneer at the "soft" sciences such as sociology, or at the humanities such as history or literature. These softer subjects cannot meet the austere standards of the logical positivists. But are they therefore without serious meaning?

Logical positivism is just too narrow. There is more meaning in everyday speech than meets the logical

positivist eye. There are more types of meaning than just verifiable (true/false) empirical or logical statements. Under the lead of Ludwig Wittgenstein philosophers began to study and analyze ordinary language and its meanings. (If you have the courage, you might like to look at Wittgenstein on meaning, here.)

Everyday Language not so bad

What is the status of a sentence like

- I will meet you at the show.

This is a promise, and, strictly speaking it has no truth value. It is not verifiable at the time it is made. Logical positivists would say it has no meaning.

John Austin would say that it does have a meaning provided you have the means to carry out your promise, and provided there is a reasonable possibility that the person to whom the message is addressed can also get to the show. You could contrast this sentence with a similar one,

- I will meet you on Mars.

This one is a truly meaningless gesture.

Austin sees such sentences as invoking a "performance" of some kind, in other words, invoking actions. He called sentences like this, involving promises, commands, requests and the like, ***performatives***. They later came to be called speech acts. Both terms have been taken over by computer scientists to describe agent communications.

John Searle, a student of Austin fully developed a theory of speech acts.

Some Notes on Searle's Speech Act Theory

Searle describes speech acts (also now know as communicative acts) in terms of structure and process. Although a speech act is intuitively simple, Searle's analysis is quite detailed. (Analysis always is. Consider analyzing the motion of a baseball pitche when he throws a fast ball.) Here we just highlight a few points. (Whole books have been written on this topic!)

A speech act can be structured as a tree.

- illocutionary act
  - illocutionary force
  - propositional act
    - referring act
    - predicating act

An illocutionary act is just another name for speech act (using Latin :-)).

For a communication to properly take place, all the component of the speech ("illocutionarty") act must be present.

Here are Searle's own examples used to illustrate the above tree structure.

1. Sam smokes habitually.
2. Does Sam smoke habitually?
3. Sam, smoke habitually! [An unlikely order.]
4. Would that Sam smoked habitually. [an unlikely wish.]

In all four of these sentences the referring act is the same: Sam.

The predicating act is also the same in each: smoking habitually.

Taken together, the referring content and the predicating content compose the propositional act. The **propositional act tells us what is being talked** about, namely, Sam's smoking habits.

But **what is being said about the subject being talked about?** This is where the illocutionary force comes in. In the above sentences the illocutionary force is implemented using word order, a common means in English. Punctuation is also used.

So in #1 we have an assertion. This will become an INFORM perforaative in the Agent Communication Language (ACL).

In #2 we have a question, which is either a QUERY-IF or QUERY-REF in ACL.

In #3 we have a command, which becomes REQUEST in ACL (because computer people are so polite).

#4 is a wish. At present, artificial agents are not capable of wishing.

In CS the illocutionary force is usually called a **performative.**

Speech Acts

(This page summarizes ideas from chapter 22 of Artificial Intelligence, a Modern Approach by Stuart Russell and Peter Norvig.)

"***In general, communication is the intentional exchange of information brought about by the production and perception of signs drawn from a shared system of conventional signs.***"

Speech Acts in Nature

Conventional Signs or Signals

Vervet monkeys

Foraging. A special loud bark from one. All head for trees. They have been warned of a stalking leopard.

These monkeys have other signals as well.Different calls for different predators (short cough for eagles, chatter for snakes), and for other activities such as different grunts depending on whether the other monkey is dominant or inferior

Humans

Also use signs. For example, smiles, frowns, hand shakes. But mainly humans use a complex ***system*** of signs and signifiers called language.

Language allows humans to communicate "an ***unbounded*** number of qualitatively different messages."

Communication as action

One action an agent can perform is to produce language. Because the original study of these acts came from linguistics, they are called speech acts. (Here "speech" is used as in the phrase "free speech". That is, typing, sky-writing and sign language, for example, are acts of speech.)

Terms used. speaker, hearer, utterance, words.

Advantages of speech acts

They allow group coordination, to the advantage of the group as a whole. They

- inform
- query
- answer
- request or command
- acknowledge
- promise and offer

- share

The hard part of acts of speech is deciding what to say. This is a complex planning problem. rather like game playing: I move (i.e., say something) anticipating your reply, and perhaps my reply to your reply, etc. The philosopher Ludwig Witgenstein spoke of "language games".

The Component Steps of communication

A typical communication episode might be: Speaker S wants to tell hearer H about proposition P using words W. There are **seven steps.**

Three steps take place in the speaker.

- **Intention**: S wants H to believe P (where S typically also believes P)
- **Generation:** S chooses the words W because they express the meaning P.
- **Synthesis**: S utters the words W(usually addressing them to H).

Four steps take place in the hearer:

- **Perception**: H hears W* (ideally W* = W but there can be misperception).
- **Analysis**: H hears the W* has possible meanings, P_1 ... P_n (words and phrases can have several meanings).
- **Disambiguation**: H infers the S intended the meaning P_i (where ideally P_i = P, but misinterpretation is possible).
- **Incorporation**: H decides to believe P_i (or rejects it as out of line with already held beliefs).

Note that analysis has two parts, parsing and semantic analysis.

Two Models of Communication

- encoded messages.
- situated language

Encoded messages are like Morse Code or some other simple transport mechanism. There is no context to the messages. The message "I am here now." by itself does not mean much if it comes over the Internet.

Situated language recognizes that context matters. For example, "I am here now." has different meanings for Bob in Toronto, and Peter in Vancouver.

Two types of communicating agents

- Agents with the same internal knowledge representation (KR) scheme
- Agents which make no assumptions about one another's internal knowledge representation

schemes. They share a common communication language.

Agents which share a common internal representation.

In human terms this kind of communication, if it were possible, would be called mental telepathy! In the case of Aglets, we would have a situation in which aglets could invoke one another's public methods.

Communication could be carried out with two generic methods Tell(KB, P) and Ask(KB, Q). The diagram illustrates.

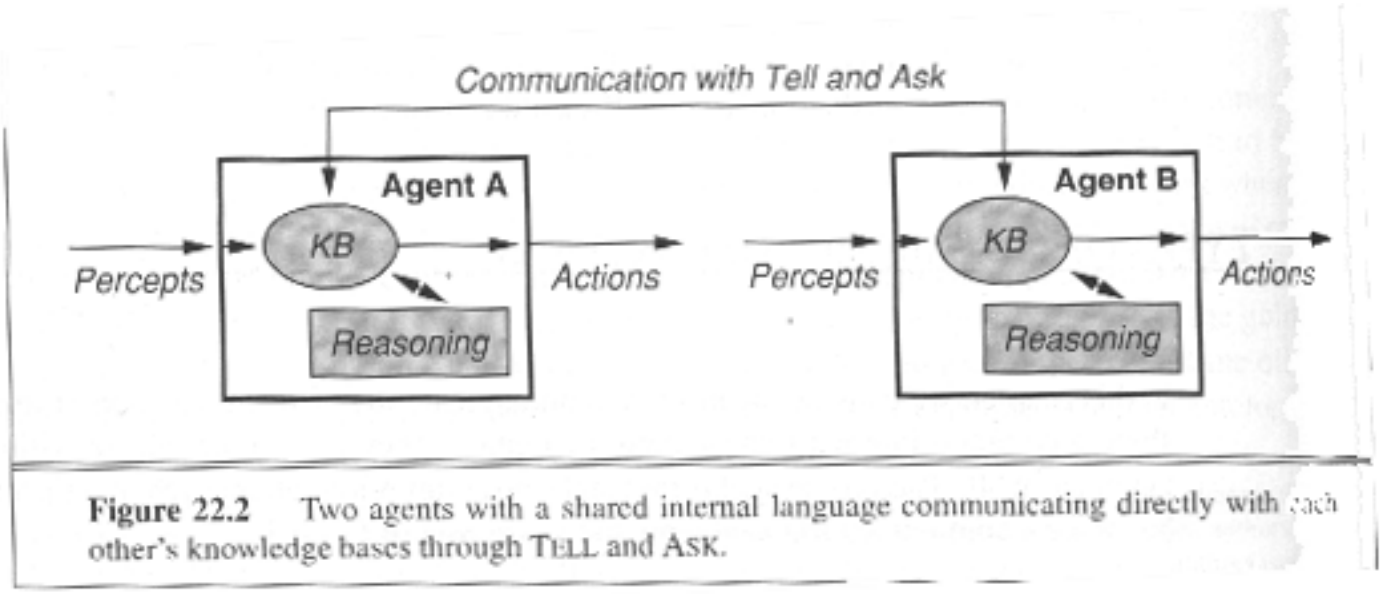


Figure 22.2 Two agents with a shared internal language communicating directly with each other's knowledge bases through TELL and ASK.

The fact that the two KB's are assumed the same means that they not only have the same structure, but that they have the same names for the objects therein. What happens, if the agents, independently moving around their respective environments, learn new things, and ***name them differently***? This is a major weakness of this communication mechanism. Language is necessary.

In other words, with this system, assuming independent learning, the ontologies of the communicating agents might be very hard to synchronize.

Communication using a common external language

This is the way humans do it, and considerable progress has been made in creating artificial agents which understand so-called natural (human) language. Still, artificial agents usually use some formalized subset of a natural language, or an artificial formal language.

This external communication language can be different from the internal representation languages of the communicating agents. Each agent can have a different internal representation language. This situation is illustrated in the following diagram.

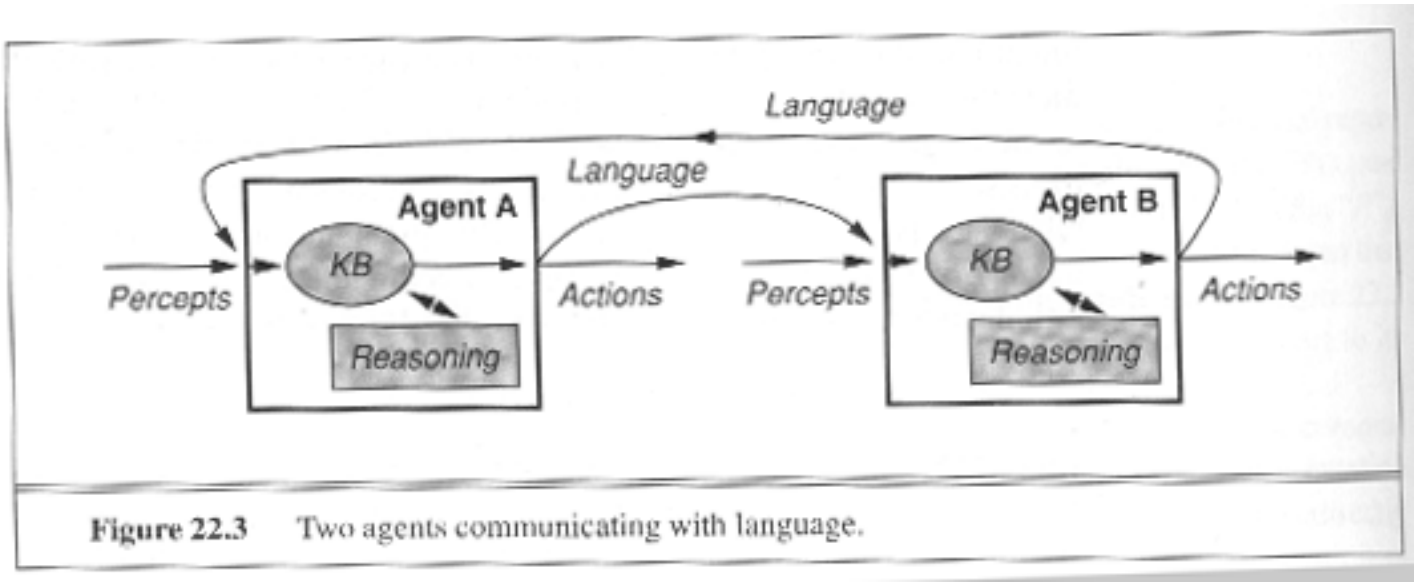


Figure 22.3 Two agents communicating with language.

Using an external communication language is clearly more flexible, but more complicated, than shared internal representation method. Especially for autonomous agents that learn, an external communication language is necessary.

Research in the last few years has developed a language to implement a set of standard, common and useful, speech acts. The earlies such language to gain fame was is KQML, the Knowledge Query and Manipulation Language together with KIF, the knowledge interchange format.

The Foundation for Intelligent Physical Agents (FIPA) is a consortium or corporations and universities from all over the world whose aim is to standardize agent communication. It was founded in 1996. Currently the two main communication standars are the Agent Communication Language (ACL) and the Semantic Language (SL), but there are other contenders as well. FIPA also constructs standards for many other aspects of agent systems.

A Summary of the Communication Act Notes

We can pull together some of the basic ideas of the previous pages this way.

Human and other Languages

Human languages differ from those of other species such as vervlet monkeys in that human languages are unbounded, whereas other species have only a finite set of calls or guestures.

Aglets have a very limited language factility. Aglet messages are just strings. You can make up a large number of message types but each just represents one idea, much like the calls of vevlet monkeys. For sophisticated communication, a more flexible mechanism is necessary.

Why hava an External Communcation Language?

Many people believe in mental telepathy or mind reading. This diagram illustrates:

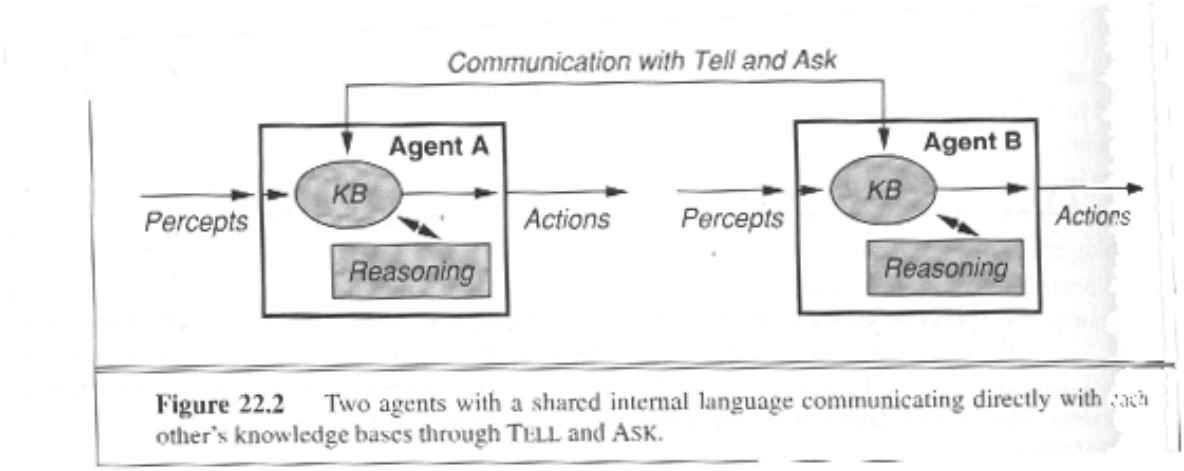


Figure 22.2 Two agents with a shared internal language communicating directly with each other's knowledge bases through TELL and ASK.

The trouble with this method, quite apart formt he lack of a transport medium (bran waves?) is that it requires too much knowledge of the internal behaviour of the other agent's brain. The two KB's would have to be the same, or very similary, an unlikely happening with autonous agents.

In the computer world (e.g., aglets) this situation correponds to each agent calling the methods of the others. (As in Sayit and HearIt aglets 1 and 2). For most cases doing this destroys agent autonomy and requires too much knowledge of the insides of the agents.

So, for hunan,animal and computer agents it is better to have an external communication language as illustrated in the following diagram:

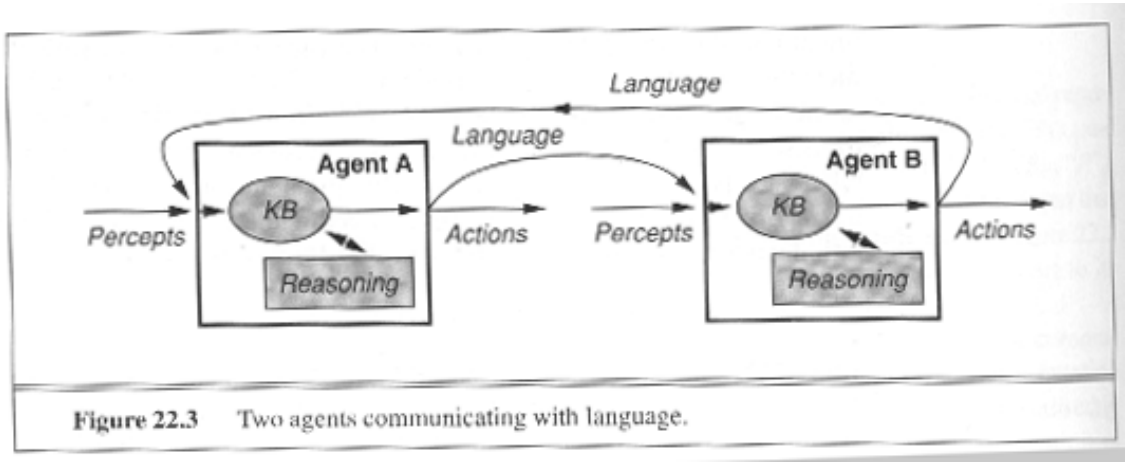


Figure 22.3 Two agents communicating with language.

With an external communication language, no knowledge of the inner workings of the agents is required.

The Question of Meaning

Communication is meant to convey meaning among agents. But when is a statement meaningful? Logical Positivists had a very strict measure of meaning. Any meainigful statement had to satisfy the "principle of verification". This meant testing it logically to test its internal consistency, and/or verifying it experimentally. Meaning was associated with verifyable truth.

The only human endeavour that fully meets this standard is science. The lanugage of science, mathematics, is indeed very powerful and it is nice if a subject lends itself to precise mathematical descriptions. Computer languages such as C or Java also have a well defined structure and no doubt would be approved of by the Logical Positivists. But, the world is complex. In many situations we are not always so lucky as to have these powerful formalisms availble.

Meaning in Everyday Language

Lewis Carrol (Alice in Wonderland etc) on meaning.

What does this mean? Jabberwocky

From the above site we have the orginal Jabberwocky poem and a "corrected" version created with the help of a computer spell checker.

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!
He took his vorpal sword in hand:
Long time the manxome foe he sought—
So rested he by the Tumtum tree,
And stood awhile in thought.

And as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!
One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.
And hast thou slain the Jabberwock?
Come to my arms, my beamish boy!

TABLESPOONS

Teas Willis, and the sticky tours
Did gym and Gibbs in the wake.
All mimes were the borrowers,
And the moderate Belgrade.

"Beware the tablespoon my son,
The teeth that bite, the Claus that catch.
Beware the Subjects bird, and shred
The serious Bandwidth!"

He took his Verbal sword in hand:
Long time the monitors fog he sought,
So rested he by the Tumbled tree,
And stood a while in thought.

And as in selfish thought he stood,
The tablespoon, with eyes of Flame,
Came stifling through the trigger wood,
And troubled as it came!

One, two! One, two! And through and though,
The Verbal blade went thicker shade.

O frabjous day! Callooh! Callay!
He chortled in his joy.

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

He left it dead, and with its head,
He went gambling back.

"And host Thai slash the tablespoon?
Come to my arms my bearish boy.
Oh various day! Cartoon! Cathay!"
He charted in his joy.

Teas Willis, and the sticky tours
Did gym and Gibbs in the wake.
All mimes were the borrowers,
And the moderate Belgrade.

Note that the original text, although full of apparently meaningless words still tells a story which human readers can decode with a little imagination. On the other hand, the spell checker text, although it contains nothing but meaningful words, is complete nonsense.

The human brain has amazing abilities to dig meaning out of language. It will be a long time before computers will compare with this intelligence.

More nonsense

What do words mean?

Ludwig Wittgenstein, John Austin and John Searle investigated the nature of meaning in everyday language. The wished to analyse the what makes a sentence such as "I'll meet you tomorrow at the show." and "I'll meet you tomorrow on Mars.". Their study lead to the idea of communication as a performace, as an action. For them speech is action.

Communicative Acts

Communicative (speech) acts have a tree structure of components:

- Communicative Act
  - performative (illocutionary force)
  - proposition
    - referent(s)
    - predicate

The predicate is the content of the speech act. But the meaning, what you are doing with the content, must also be considered in determining the meaing of a speech act.

You will see the structure of speech acts reflected in the agent communication languages designed by FIPA.

Agent Communication Languages

KQML

KQML, Knowledge Communication Meta Language was one of the earliest attempts ot construct an agent communicaion language based on speech act theory. It has had a major influence on later develpments.

Notes on KQML

ACL

ACL, the Agent Communication Language, is an based on KQML and represents the world standard for agent communication as proposed by FIPA. Quite a few agent systems "speak" ACL. FIPA defines a library of allowed communicative acts.

FIPA Communicative Act Library Specification for ACL

- pdf (local) version
- html version

You can see that ACL is closely related to KQML. In fact, ACL represents a smaller version of KQML which is more precidely defined.

XML

XML has become quite popular. There is a standard version of ACL written in XML and sponsored by FIPA.

FIPA xmlacl specification (DTD)

A Closer Look at Some Communicative Acts

INFORM

3.8 Inform

Summary	The sender informs the receiver that a given proposition is true.
Message Content	A proposition.
Description	The sending agent: • holds that some proposition is true,

	• intends that the receiving agent also comes to believe that the proposition is true, and, • does not already believe that the receiver has any knowledge of the truth of the proposition. The first two properties defined above are straightforward: the sending agent is sincere, and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last property is concerned with the semantic soundness of the act. If an agent knows already that some state of the world holds (that the receiver knows proposition *p*), it cannot rationally adopt an intention to bring about that state of the world (*i.e.* that the receiver *comes to know p* as a result of the inform act). Note that the property is not as strong as it perhaps appears. The sender *is not* required to *establish* whether the receiver knows p. It is only the case that, in the case that the sender already happens to know about the state of the receiver's beliefs, it should not adopt an intention to tell the receiver something it already knows. From the receiver's viewpoint, receiving an inform message entitles it to believe that: • the sender believes the proposition that is the content of the message, and, • the sender wishes the receiver to believe that proposition also. Whether or not the receiver does, indeed, adopt belief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.
Formal Model	$\langle i, \text{inform } (j, \phi) \rangle$ FP: $B_i\phi \wedge \neg B_i(\text{Bif}_j\phi \vee \text{Uif}_j\phi)$ RE: $B_j\phi$
Examples	Agent *i* informs agent *j* that (it is true that) it is raining today. (inform :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content "weather (today, raining)" :language Prolog)

In the example you can see the influence of speech act theory. The "illocutionary force", or performative is INFORM. The referents are given in the :sender and :receiver slots.

Most interesting is the :content slot. Since the purpose fo the INFORM performative is to convey presumably true information, the content of an INFORM performative is normally a predicate, which can be true or false.

In the example, content just uses strings which represent some language. This is the content language. It can be anything. The :language slot specifies which. In this example, it is Prolog. In JADE it is normally SL.

REQUEST

3.19 Request

Summary	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
Message Content	An action expression.
Description	The sender is requesting the receiver to perform some action. The content of the message is a description of the action to be performed, in some language the receiver understands. The action can be any action the receiver is capable of performing: pick up a box, book a plane flight, change a password, etc. An important use of the request act is to build composite conversations between agents, where the actions that are the object of the request act are themselves communicative acts such as *inform*.
Formal Model	$\langle i,$ request $(j, a)\rangle$ FP: FP (a) $[i\backslash j] \wedge B_i$ Agent $(j, a) \wedge \neg B_i\ I_j$ Done (a) RE: Done (a) FP(a) $[i\backslash j]$ denotes the part of the FPs of *a* which are mental attitudes of *i*.
Examples	Agent *i* requests *j* to open a file. (request :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content "open \"db.txt\" for input"

```
:language vb)
```

REQUEST asks the receiver to take some action. The action can, of course, be a speech act, often INFORM. The :content language can be a string containing anything, vene Visual Basic! Of course the receiving agent has to understand the content language.

QUERY-REF

3.16 Query Ref

Summary	The action of asking another agent for the object referred to by an referential expression.
Message Content	A descriptor (a referential expression).
Description	*Query-ref* is the act of asking another agent to inform the requester of the cbject identified by a descriptor. The sending agent is requesting the receiver to perform an *inform* act, containing the object that corresponds to the descriptor. The agent performing the *query-ref* act: • does not know which object or set of objects corresponds to the descriptor, and, • believes that the other agent can inform the querying agent the object or set of objects that correspond to the descriptor.
Formal Model	$\langle i,$ query-ref $(j,$ *Ref* $x\ \delta(x))\rangle \equiv$ $\langle i,$ request $(j, \langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle)\rangle$ FP: $\neg Bref_i(Ref\ x\ \delta(x)) \wedge \neg Uref_i(Ref\ x\ \delta(x)) \wedge$ $\neg B_i\ I_j\ Done(\langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle)$ RE: $Done(\langle i,$ inform $(j,$ *Ref* $x\ \delta(x) = r_1)\rangle\ |...|$ $\langle i,$ inform $(j,$ *Ref* $x\ \delta(x) = r_k)\rangle)$ Note: *Ref* $x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$.
Example	Agent *i* asks agent *j* for its available services. (query-ref :sender (agent-identinfier :name i)

```
  :receiver (set (agent-identifier :name j))
  :content
     ((all ?x (available-service j ?x)))
  …)
```

Agent *j* replies that it can reserve trains, planes and automobiles.

```
(inform
  :sender (agent-identifier :name j)
  :receiver (set (agent-identifier :name i))
  :content
    ((= (all ?x (available-service j ?x))
      (set (reserve-ticket train)
           (reserve-ticket plane)
           (reserve automobile))))
  …)
```

This one is a bit more complicated. The content language in the example is SL.

QUERY-REF is used to refer to objects that the receiver knows about but which the names are not known to the sender. So the sender sends a description of the required object or objects. In other words, the sender sends an expression, the value of which references the desired objects.

In the above example, the sender used the variable ?x as a reference to the objects which match the description "available services offered by agent j". Agent j responds with an INFORM message.

You can see from this example that SL is quite a tricky language. Fortunately, as of version 2.4, JADE has some helpful packages to help out.

4 KQML AS A COMMUNICATION LANGUAGE

[from D. Benaech & T. Desprats, A KQML-CORBA based Architecture for Intelligent Agents Communication in Cooperative Service and Network Management. Postscript version]

4.1 Overview of KQML specification [KQML93] [LABR96]

4.1.1 The KQML key features

KQML was conceived as both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML's key features are:

- KQML messages are opaque to the content they carry. KQML messages do not merely communicate sentences in some language, but rather communicate an attitude about the content (assertion, request, query, basic response, etc.).
- The language's primitives are called performatives (this term comes directly from the speech act theory). Performatives define the permissible actions (operations) that agents may attempt in communicating with one another.
- KQML assumes that at the agent level, the communication appears as a point- to-point message passing.
- .An environment of KQML speaking agents may be enriched with special agents, called facilitators, that provide to the agents additional functions to deal with networking (association of physical addresses with symbolic names, registration of agents and/or services offered and sought by agents, enhanced communication services as forwarding, brokering, broadcasting...).

KQML may be considered as a communication language for the exchange of information and knowledge between agents, through the use of a set of standard message types. Next is an example of KQML message.

```
(ask-if
        :sender A
        :receiver B
        :language Prolog
        :ontology foo
        :reply-with id1
        :content ``bar(a,b)" )
```

In KQML terminology, ask-if is a performative. A performative sets parameters that are introduced by keywords. In this example, the agent named A (:sender) is querying the agent B (:receiver), in Prolog (:language) about the truth status of ``bar(a,b)" (:content). Any response to this KQML message will be identified by id1 (:reply-with). The **ontology** name foo may provide additional information regarding the interpretation of the :content. Let's

suppose that B is not able to perform the action suggested by A in the previous message. B will answer to A by using the next performative.

```
(sorry
    :sender B
    :receiver A
    :in-reply-to id1
    :reply-with id2 )
```

B (:sender) uses the performative sorry to inform A(:receiver) that it cannot perform the evaluation of bar(a, b). The agent A will perfectly know that this message refers to this evaluation because B used the :in- reply-to parameter with a value of id1.

What is an ontology?

KQML Performatives

4.1.2 The KQML string syntax

A performative (i.e. a KQML message) is expressed as an ASCII string using a syntax which has been defined in a BNF. This syntax is a restriction on the ASCII representation of Common Lisp Polish-prefix notation. This notation has the advantages of being readable by humans, simple for programs to parse and transportable by many inter process messaging platforms. Parameters in performative are indexed by keywords, must begin by a colon (:) and must precede the corresponding parameter value. They are order independent. Table 1 Summary of reserved parameter keywords and their meanings

Keyword	Meaning
:sender	actual sender of the performative
:receiver	actual receiver of the performative
:reply-with	expected label in a response to the current message
:in-reply-with	expected label in a response to a previous message (same as the :reply-with value of the previous message)
:language	name of the representation language of the :content
:ontology	name of the ontology assumed in the :content
:content	information about which the performative expresses an attitude
:from	origin of the performative in :content when forward is used
:to	final destination when forward is used

4.1.3 The KQML reserved performatives

No less than 35 reserved performatives are defined in the KQML specification. We do not detail each of these performatives in this paper, but a deeper description can be found in [LABR96]. In order to give the reader an overview of the communication semantics these performatives express, a classification can be proposed to summarise these semantics:

(A) **Discourse performatives**: these are the performatives to be used in the context of an information and knowledge exchange between two agents. They may be considered as close as possible to speech acts theory.

(B) **Intervention and mechanics of conversation performatives**: their role is to intervene in the normal course of a conversation. The normal course of a conversation is as follows: agent A sends a KQML message (thus starting a conversation) and agent B responds whenever it has a response or a follow-up. The performative of this category either prematurely terminate a conversation (error, sorry) or override this default protocol (standby, ready, next, rest and discard).

(C) **Networking and Facilitation performatives** are not speech acts in the pure sense. They allow agents to find other agents that can process their queries.

4.1.4 Domain of KQML speaking agents

We now summarise the features of a domain of KQML-speaking agents. In each domain there is at least one agent with a special status called facilitator that can always handle the networking and facilitation performatives. Agents advertise to their facilitator thus announcing the messages that they are committed to accept and properly process. Advertising to a facilitator is like advertising to the community. So, agents can use their facilitator either:

- to have their queries properly dispatched to other agents, using recruit-one, recruit-all, broker-one or broker-all, or
- to send a recommend-one or a recommend-all to get the relevant advertise message and directly contact agent(s) that may process their queries.

Agents can access agents in other domain through their facilitator, or directly. The term facilitator is used to refer to all kinds of special services that may be provided by specialised agents, such as Agent Name Servers, proxy agents, traders or brokers.

4.2 Suitability of KQML

Section 3 showed the need to provide a communication language that can support two basic styles of interaction between intelligent agents. Both ``Query/Response'' and ``knowledge exchange'' interaction styles should be supported to allow the agents to exchange tasks, information and functionality. Broadcast, multicast, group and location facilities also need to be provided in order to support multi-agents interaction. The capacity of KQML to cover these requirements appears in the following properties:

- KQML offers a range of reserved performatives to allow an agent to send queries (ask-if, ask-one, ask-all, achieve...) to another one.
- It also proposes performatives to permit an agent to reply to another one (tell, eos, sorry...).
- Other performatives are able to support generic information exchange (tell, untell, deny...), functionality transfer (insert, tell...) and capability definition (advertise, subscribe...).
- An important number of reserved performatives are concerned by networking and group facilities: register, forward, broadcast, recommend, broker...
- the :content parameter of a performative is an opaque message. This constitutes an important benefit that KQML may bring to improve

interoperability. The nature of the :content parameter can vary for example from a SNMP get request, to a KQML performative or a CMIP notification.

These characteristics of KQML made us believe that this language should be a suitable support for the communication between intelligent agents in the specific context of cooperative service and NM. Finest analysis and practical studies are now necessary to answer to some issues:

- Are all the KQML reserved performatives indispensable in the cooperative NM context?
- Since KQML is still an open and evolving specification language (you may add yours own performatives with respect to the specifications), it might be attractive to define a specific set of performatives dedicated to the particular interactions between NM agents.
- Is an implementation of KQML easily realisable? What is the most suitable/efficient transport protocol to support all the networking facilities proposed by KQML?

We started to study the last issue by developing a KQML implementation based on CORBA. The next section gives an overview of a KQML-CORBA based architecture able to support cooperative services and NM applications.

What is an Ontology?

Tom Gruber <gruber@ksl.stanford.edu>

Short answer:

> An ontology is a specification of a conceptualization.

The word "ontology" seems to generate a lot of controversy in discussions about AI. It has a long history in philosophy, in which it refers to the subject of existence. It is also often confused with epistemology, which is about knowledge and knowing.

In the context of knowledge sharing, I use the term ontology to mean a *specification of a conceptualization*. That is, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set-of-concept-definitions, but more general. And it is certainly a different sense of the word than its use in philosophy.

What is important is what an ontology is *for*. My colleagues and I have been designing ontologies for the purpose of enabling knowledge sharing and reuse. In that context, an ontology is a specification used for making ontological commitments. The formal definition of ontological commitment is given below. For pragmetic reasons, we choose to write an ontology as a set of definitions of formal vocabulary. Although this isn't the only way to specify a conceptualization, it has some nice properties for knowledge sharing among AI software (e.g., semantics independent of reader and context). Practically, an ontological commitment is an agreement to use a vocabulary (i.e., ask queries and make assertions) in a way that is consistent (but not complete) with respect to the theory specified by an ontology. We build agents that commit to ontologies. We design ontologies so we can share knowledge with and among these agents.

This definition is given in the article:

T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199-220, 1993. Available on line.

A more detailed description is given in

T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Presented at the Padua workshop on Formal Ontology, March 1993, to appear in an edited collection by Nicola Guarino. Available online.

With an excerpt attached.

Ontologies as a specification mechanism

A body of formally represented knowledge is based on a *conceptualization*: the objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them (Genesereth & Nilsson, 1987) . A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. Every knowledge base, knowledge-based system, or

knowledge-level agent is committed to some conceptualization, explicitly or implicitly.

An **ontology** is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what "exists" is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge. Thus, in the context of AI, we can describe the ontology of a program by defining a set of representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. Formally, an ontology is the statement of a logical theory.[1]

We use common ontologies to describe *ontological commitments* for a set of agents so that they can communicate about a domain of discourse without necessarily operating on a globally shared theory. We say that an agent **commits** to an ontology if its observable actions are consistent with the definitions in the ontology. The idea of ontological commitments is based on the Knowledge-Level perspective (Newell, 1982) . The Knowledge Level is a level of description of the knowledge of an agent that is independent of the symbol-level representation used internally by the agent. Knowledge is attributed to agents by observing their actions; an agent "knows" something if it acts *as if* it had the information and is acting rationally to achieve its goals. The "actions" of agents---including knowledge base servers and knowledge-based systems--- can be seen through a tell and ask functional interface (Levesque, 1984) , where a client interacts with an agent by making logical assertions (tell), and posing queries (ask).

Pragmatically, a common ontology defines the vocabulary with which queries and assertions are exchanged among agents. Ontological commitments are agreements to use the shared vocabulary in a coherent and consistent manner. The agents sharing a vocabulary need not share a knowledge base; each knows things the other does not, and an agent that commits to an ontology is not required to answer all queries that can be formulated in the shared vocabulary.

In short, a commitment to a common ontology is a guarantee of consistency, but not completeness, with respect to queries and assertions using the vocabulary defined in the ontology.

Notes

[1] Ontologies are often equated with taxonomic hierarchies of classes, but class definitions, and the subsumption relation, but ontologies need not be limited to these forms. Ontologies are also not limited to conservative definitions, that is, definitions in the traditional logic sense that only introduce terminology and do not add any knowledge about the world (Enderton, 1972) . To specify a conceptualization one needs to state axioms that do constrain the possible interpretations for the defined terms.

Name	*Section*	*Meaning*
achieve	5.6	S wants R to do make something true of their environment
advertise	5.8	S is particulary-suited to processing a performative
ask-about	5.4	S wants all relevant sentences in R's VKB
ask-all	5.4	S wants all of R's answers to a question
ask-if	5.4	S wants to know if the sentence is in R's VKB
ask-one	5.4	S wants one of R's answers to a question
break	5.10	S wants R to break an established pipe
broadcast	5.10	S wants R to send a performative over all connections
broker-all	5.11	S wants R to collect all responses to a performative
broker-one	5.11	S wants R to get help in responding to a performative
deny	5.1	the embedded performative does not apply to S (anymore)
delete	5.2	S wants R to remove a ground sentence from its VKB
delete-all	5.2	S wants R to remove all matching sentences from its VKB
delete-one	5.2	S wants R to remove om matching sentence from its VKB
discard	5.7	S will not want R's remaining responses to a previous performative
eos	5.5	end of a stream of responses to an earlier query
error	5.3	S considers R's ealier message to be mal-formed
evaluate	5.4	S wants R to simplify the sentence
forward	5.10	S wants R to route a performative
generator	5.7	same as a standby of a stream-all
insert	5.2	S asks R to add content to its VKB
monitor	5.9	S wants updates to R's response to a stream-all
next	5.7	S wants R's next response to a previously-mentioned performative
pipe	5.10	S wants R to route all further performatives to a another agent
ready	5.7	S is ready to respond to R's previously-mentioned performative
recommend-all	5.11	S wants all names of agents who can respond to a performative
recommend-one	5.11	S wants the name of an agent who can respond to a performative
recruit-all	5.11	S wants R to get all suitable agents to respond to a performative
recruit-one	5.11	S wants R to get another agent to respond to a performative
register	5.10	S can deliver performatives to some named agent
reply	5.4	communicates an expected reply
rest	5.7	S wants R's remaining responses to a previously-mentioned performative
sorry	5.3	S cannot provide a more informative reply
standby	5.7	S wants R to be ready to respond to a performative
stream-about	5.5	multiple-response version of ask-about
stream-all	5.5	multiple-response version of ask-all
subscribe	5.9	S wants updates to R's response to a performative
tell	5.1	the sentence in S's VKB
transport-address	5.10	S associates symbolic name with transport address
unregister	5.10	a deny of a register
untell	5.1	the sentence is not in S's VKB

Table 2: Summary of reserved performatives, for sender S and recipient R.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA Communicative Act Library Specification

Document title	FIPA Communicative Act Library Specification		
Document number	XC00037G	**Document source**	FIPA TC C
Document status	Experimental	**Date of this status**	2001/01/29
Supersedes	FIPA00003, FIPA00038, FIPA00039, FIPA00040, FIPA00041, FIPA00042, FIPA00043, FIPA00044, FIPA00045, FIPA00046, FIPA00047, FIPA00048, FIPA00049, FIPA00050, FIPA00051, FIPA00052, FIPA00053, FIPA00054, FIPA00055, FIPA00056, FIPA00057, FIPA00058, FIPA00059, FIPA00060		
Contact	fab@fipa.org		
Change history			
2001/01/29	Approved for Experimental		

Geneva, Switzerland

Foreword

The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. FIPA makes the results of its activities available to all interested parties and intends to contribute its results to the appropriate formal standards bodies.

The members of FIPA are individually and collectively committed to open competition in the development of agent-based applications, services and equipment. Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international organization without restriction. In particular, members are not bound to implement or use specific agent-based standards, recommendations and FIPA specifications by virtue of their participation in FIPA.

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Procedures for Technical Work. A complete overview of the FIPA specifications and their current status may be found in the FIPA List of Specifications. A list of terms and abbreviations used in the FIPA specifications may be found in the FIPA Glossary.

FIPA is a non-profit association registered in Geneva, Switzerland. As of January 2000, the 56 members of FIPA represented 17 countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found at http://www.fipa.org/.

Contents

1 Introduction

This document contains specifications for structuring the FIPA Communicative Act Library (FIPA CAL) including: status of a FIPA-compliant communicative act, maintenance of the library and inclusion criteria.

This document is primarily concerned with defining the structure of the FIPA CAL and the requirements for a proposed communicative act to be included in the library. The elements of the library are listed in this document.

This document also contains the formal basis of FIPA ACL semantics in the annex for the semantic characterization of each FIPA communicative act.

2 Overview

This document focuses on the organization, structure and status of the FIPA Communicative Act Library, FIPA CAL and discusses the main requirements that a communicative act must satisfy in order to be FIPA-compliant.

The objectives of standardizing and defining a library of FIPA compliant communicative acts are:

- To help ensure interoperability by providing a standard set of composite and macro communicative acts, derived from the FIPA primitive communicative acts,
- To facilitate the reuse of composite and macro communicative acts, and,
- To provide a well-defined process for maintaining a set of communicative acts and act labels for use in the FIPA ACL.

In the following, we present the basic principles of the FIPA CAL. These principles help to guarantee that the CAL is stable, that there are public rules for the inclusion and maintenance of the CAL and that developers seeking communicative acts for their applications can use the CAL.

2.1 Status of a FIPA-Compliant Communicative Act

The definition of a communicative act belonging to the FIPA CAL is normative. That is, if a given agent implements one of the acts in the FIPA CAL, then it must implement that act in accordance with the semantic definition in the FIPA CAL. However, FIPA-compliant agents are not required to implement any of the FIPA CAL languages, except the not-understood composite act.

By collecting communicative act definitions in a single, publicly accessible registry, the FIPA CAL facilitates the use of standardized Communicative Acts by agents developed in different contexts. It also provides a greater incentive to developers to make any privately developed communicative acts generally available.

The name assigned to a proposed communicative act must uniquely identify which communicative act is used within a FIPA ACL message. It must not conflict with any names currently in the library, and must be an English word or abbreviation that is suggestive of the semantics. The FIPA Agent Communication Technical Committee is the initial judge of the suitability of a name.

FIPA is responsible for maintaining a consistent list of approved and proposed communicative act names and for making this list publicly available to FIPA members and non-members. This list is derived from the FIPA Communicative Act Library.

In addition to the semantic characterization and descriptive information that is required, each Communicative Act in the FIPA CAL may specify additional information, such as stability information, versioning, contact information, different support levels, etc.

2.2 FIPA Communicative Act Library Maintenance

The most effective way of maintaining the FIPA Communicative Act Library is through the use of the communicative acts themselves by different agent developers. This is the most direct way of discovering possible bugs, errors, inconsistencies, weaknesses, possible improvements, as well as capabilities, strengths, efficiency etc. In order to collect feedback on the communicative acts in the library and to promote further research, FIPA encourages coordination between agent language designers, agent developers, and FIPA members.

FIPA will designate a Technical Committee to maintain the FIPA CAL. The FIPA CAL will be managed by this technical committee, which will be responsible for the following items:

- Collecting feedback and the comments about communicative acts in the FIPA CAL. Depending on interest, the technical committee may organize more specific Working Groups. These groups would be responsible for maintaining public lists referring to projects and people who are currently working on different communicative acts.

- Inviting contributions in various forms: e-mail comments, written reports, papers, technical documents, and so forth. The current email address of the technical committee is specified on the first page of this document.

- All technical committee members will be notified about contributions, comments or proposed changes and should be able to access them.

- The proposed updates to the FIPA CAL must be discussed and approved during an official FIPA meeting, in order that the FIPA community may be involved with and informed of all of the FIPA approved communicative acts in the library

- In the future, FIPA intends to supply templates (publicly accessible from the FIPA web site) in order to facilitate submission of candidate communicative acts to the FIPA CAL, and to ensure that agent language developers understand and can easily satisfy the requirements for the submission of a new communicative act to the FIPA CAL.

2.3 Inclusion Criteria

In order to populate the FIPA CAL, it is necessary to set some fundamental guidelines for the selection of specific communicative acts.

The minimal criteria that must be satisfied for a communicative act to be included in the FIPA CAL are:

- A summary of the candidate act's semantic force and content type are required.

- A detailed natural language description of the act and its consequences are required.

- A formal model, written in SL, of the act's semantics, its formal preconditions, and its rational effects is required.

- Examples of the usage of the new communicative act are required.

- Substantial and clear documentation must be provided. This means that the proposal must be already well structured. FIPA members are in no way responsible for translating submitted communicative acts into an acceptable form. See the form of the acts in the library for a sample.

- The utility of such a new communicative act should be made clear. In particular, it should be clear that the need it solves is reasonably general, and that this need would be cumbersome to meet by combining existing communicative acts.

FIPA does not enforce the use of any particular communicative act, except for the case of *not-understood*, and those acts which are required to meet the agent management needs of the agent.

3 FIPA Communicative Acts

3.1 Accept Proposal

Summary	The action of accepting a previously submitted proposal to perform an action.
Message Content	A tuple consisting of an action expression denoting the action to be done, and a proposition giving the conditions of the agreement.
Description	*Accept-proposal* is a general-purpose acceptance of a proposal that was previously submitted (typically through a *propose* act). The agent sending the acceptance informs the receiver that it intends that (at some point in the future) the receiving agent will perform the action, once the given precondition is, or becomes, true. The proposition given as part of the acceptance indicates the preconditions that the agent is attaching to the acceptance. A typical use of this is to finalize the details of a deal in some protocol. For example, a previous offer to "hold a meeting anytime on Tuesday" might be accepted with an additional condition that the time of the meeting is 11.00. Note for future extension: an agent may intend that an action become done without necessarily intending the precondition. For example, during negotiation about a given task, the negotiating parties may not unequivocally intend their opening bids: agent *a* may bid a price *p* as a precondition, but be prepared to accept price *p'*.
Formal Model	$\langle i,$ accept-proposal $(j, \langle j, act\rangle, \phi)\rangle \equiv$ $\langle i,$ inform $(j, I_i$ Done $(\langle j, act\rangle, \phi))\rangle$ FP: $B_i\ \alpha \wedge \neg B_i$ (Bif$_j\ \alpha \vee$ Uif$_j\ \alpha$) RE: $B_j\ \alpha$ Where: $\alpha = I_i$ Done $(\langle j, act\rangle, \phi)$

Example	Agent *i* informs *j* that it accepts an offer from *j* to stream a given multimedia title to channel 19 when the customer is ready. Agent *i* will inform *j* of this fact when appropriate.
	
	`(accept-proposal`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :in-reply-to bid089`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (stream-content movie1234 19))`
	`  (B (agent-identifier :name j)`
	`  (ready customer78)))`
	`  :language FIPA-SL)`

3.2 Agree

Summary	The action of agreeing to perform some action, possibly in the future.
Message Content	A tuple, consisting of an action expression denoting the action to be done, and a proposition giving the conditions of the agreement.
Description	*Agree* is a general-purpose agreement to a previously submitted *request* to perform some action. The agent sending the agreement informs the receiver that it does intend to perform the action, but not until the given precondition is true. The proposition given as part of the *agree* act indicates the qualifiers, if any, that the agent is attaching to the agreement. This might be used, for example, to inform the receiver when the agent will execute the action which it is agreeing to perform. Pragmatic note: The precondition on the action being agreed to can include the perlocutionary effect of some other CA, such as an *inform* act. When the recipient of the agreement (for example, a contract manager) wants the agreed action to be performed, it should then bring about the precondition by performing the necessary CA. This mechanism can be used to ensure that the contractor defers performing the action until the manager is ready for the action to be done.
Formal Model	$\langle i, \text{agree}\ (j, \langle i, act\rangle, \phi)\rangle \equiv$ $\langle i, \text{inform}\ (j, I_i\ \text{Done}\ (\langle i, act\rangle, \phi))\rangle$ FP: $B_i\,\alpha \wedge \neg B_i\,(\text{Bif}_j\,\alpha \vee \text{Uif}_j\,\alpha)$ RE: $B_j\,\alpha$ Where: $\alpha = I_i\ \text{Done}(\langle i, act\rangle, \phi)$ Note that the formal difference between the semantics of *agree* and the semantics of *accept-proposal* rests on which agent is performing the action.

Example	Agent *i* (a job-shop scheduler) requests *j* (a robot) to deliver a box to a certain location. *J* answers that it agrees to the request but it has low priority.
	
	`(request`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (deliver box017 (loc 12 19))))`
	`  :protocol fipa-request`
	`  :language FIPA-SL`
	`  :reply-with order567)`
	
	`(agree`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (deliver box017 (loc 12 19)))`
	`  (priority order567 low))`
	`  :in-reply-to order567`
	`  :protocol fipa-request`
	`  :language FIPA-SL)`

3.3 Cancel

Summary	The action of one agent informing another agent that the first agent no longer has the intention that the second agent perform some action.
Message Content	An action expression denoting the action that is no longer intended.
Description	*Cancel* allows an agent *i* to inform another agent *j* that *i* no longer intends that *j* perform a previously requested action. This is not the same as *i* informing *j* that *i* intends that *j* not perform the action or stop performing an action. *Cancel* is simply used to let an agent know that another agent no longer has a particular intention. (In order for *i* to stop *j* from performing an action, *i* should *request* that *j* stop that action. Of course, nothing in the ACL semantics guarantees that *j* will actually stop performing the action; *j* is free to ignore *i*'s request.) Finally, note that the action that is the object of the act of cancellation should be believed by the sender to be ongoing or to be planned but not yet executed.
Formal Model	$\langle i, \text{cancel } (j, a)\rangle \equiv$ $\langle i, \text{disconfirm } (j, I_i \text{ Done } (a))\rangle$ FP: $\neg I_i \text{ Done } (a) \wedge B_i (B_j I_i \text{ Done } (a) \vee U_j I_i \text{ Done } (a))$ RE: $B_j \neg I_i \text{ Done } (a)$ *Cancel* applies to any form of *request*ed action. Suppose an agent *i* has requested an agent *j* to perform some action *a*, possibly if some condition holds. This request has the effect of *i* informing *j* that *i* has an intention that *j* perform the action *a*. When *i* comes to drop its intention, it can inform *j* that it no longer has this intention with a *disconfirm*.
Example	Agent *j* asks *i* to cancel a previous *request-whenever* by quoting the action. (cancel :sender (agent-identifier :name j) :receiver (set (agent-identifier :name i)) :content ((action (agent-identifier :name j) (request-whenever :sender (agent-identifier :name j) :receiver (set(agent-identifier :name i)) :content[1] "((action (agent-identifier :name i) (inform-ref :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content[2] \"((iota ?x (=(price widget) ?x))\") (> (price widget) 50))" …))) :langage FIPA-SL …)

[1] The *request-whenever* message's `:content` parameter in the context of the *cancel* message is an embedded action expression. So, since this example uses SL as a content language, the content tuple of the *request-whenever* message must be converted into a Term of SL.

[2] The content of this *inform-ref* is further embedded in an embedded request-whenever message's content. So, because this example uses SL as a content language, the quote mark is itself escaped by '\'.

3.4 Call for Proposal

Summary	The action of calling for proposals to perform a given action.
Message Content	A tuple containing an action expression denoting the action to be done, and a referential expression defining a single-parameter proposition which gives the preconditions on the action.
Description	*CFP* is a general-purpose action to initiate a negotiation process by making a call for proposals to perform the given action. The actual protocol under which the negotiation process is established is known either by prior agreement, or is explicitly stated in the *:protocol* parameter of the message. In normal usage, the agent responding to a *cfp* should answer with a proposition giving the value of the parameter in the original precondition expression (see the statement of *cfp*'s rational effect). For example, the *cfp* might seek proposals for a journey from Frankfurt to Munich, with a condition that the mode of travel is by train. A compatible proposal in reply would be for the 10.45 express train. An incompatible proposal would be to travel by airplane. Note that *cfp* can also be used to simply check the availability of an agent to perform some action. Also note that this formalization of *cfp* is restricted to the common case of proposals characterized by a single parameter (x) in the proposal expression. Other scenarios might involve multiple proposal parameters, demand curves, free-form responses, and so forth.
Formal Model	$\langle i, \text{cfp}\ (j, \langle j, act\rangle, Ref\ x\ \phi(x))\rangle \equiv$ $\langle i, \text{query-ref}\ (j, Ref\ x\ (I_i\ \text{Done}\ (\langle j, act\rangle, \phi(x)) \Rightarrow$ $(I_j\ \text{Done}\ (\langle j, act\rangle, \phi(x))))\rangle$ FP: $\neg \text{Bref}_i(Ref\ x\ \alpha(x)) \wedge \neg \text{Uref}_i(Ref\ x\ \alpha(x)) \wedge$ $\neg B_i\ I_j\ \text{Done}\ (\langle j, \text{inform-ref}\ (i, Ref\ x\ \alpha(x))\rangle)$ RE: Done $(\langle j, \text{inform}\ (i, Ref\ x\ \alpha(x) = r_1)\rangle \mid \ldots \mid$ $\langle j, \text{inform}\ (i, Ref\ x\ \alpha(x) = r_k)\rangle)$ Where: $\alpha(x) = I_i\ \text{Done}\ (\langle j, act\rangle, \phi(x)) \Rightarrow I_j\ \text{Done}\ (\langle j, act\rangle, \phi(x))$ Agent *i* asks agent *j*: "What is the '*x*' such that you will perform action *'act'* when '$\phi(x)$' holds?" Note: $Ref\ x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$. Note: The RE of this is not a proposal by the recipient. Rather, it is the value of the proposal parameter. See the example in the definition of the *propose* act.

Example	Agent *j* asks *i* to submit its proposal to sell 50 boxes of plums.
	
	`(cfp`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((action (agent-identifier :name i)`
	`  (sell plum 50))`
	`  (any ?x (and (= (price plum) ?x) (< ?x 10))))`
	`  :ontology fruit-market)`

3.5 Confirm

Summary	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.
Message Content	A proposition.
Description	The sending agent: • believes that some proposition is true, • intends that the receiving agent also comes to believe that the proposition is true, and, • believes that the receiver is *uncertain* of the truth of the proposition. The first two properties defined above are straightforward: the sending agent is sincere[3], and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last pre-condition determines when the agent should use *confirm* vs. *inform* vs. *disconfirm*: *confirm* is used precisely when the other agent is already known to be uncertain about the proposition (rather than *uncertain* about the negation of the proposition). From the receiver's viewpoint, receiving a *confirm* message entitles it to believe that: • the sender believes the proposition that is the content of the message, and, • the sender wishes the receiver to believe that proposition also. Whether or not the receiver does, indeed, change its mental attitude to one of belief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.
Formal Model	$\langle i$, confirm (j, ϕ)$\rangle$ FP: $B_i\phi \wedge B_i U_j \phi$ RE: $B_j\phi$

Examples	Agent *i* confirms to agent *j* that it is, in fact, true that it is snowing today.
	`(confirm`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  "weather (today, snowing)"`
	`  :language Prolog)`

[3] Arguably there are situations where an agent might not want to be sincere, for example to protect confidential information. We consider these cases to be beyond the current scope of this specification.

3.6 Disconfirm

Summary	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.
Message Content	A proposition.
Description	The *disconfirm* act is used when the agent wishes to alter the known mental attitude of another agent. The sending agent: • believes that some proposition is false, • intends that the receiving agent also comes to believe that the proposition is false, and, • believes that the receiver either believes the proposition, or is *uncertain* of the proposition. The first two properties defined above are straightforward: the sending agent is sincere[3], and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last pre-condition determines when the agent should use *confirm* vs. *inform* vs. *disconfirm*: *disconfirm* is used precisely when the other agent is already known to believe the proposition or to be uncertain about it. From the receiver's viewpoint, receiving a *disconfirm* message entitles it to believe that: • the sender believes that the proposition that is the content of the message is false, and, • the sender wishes the receiver to believe the negated proposition also. Whether or not the receiver does, indeed, change its mental attitude to one of disbelief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.
Formal Model	$\langle i,$ disconfirm $(j, \phi)\rangle$ FP: $B_i \neg\phi \wedge B_i(U_j\phi \vee B_j\phi)$ RE: $B_j \neg\phi$

Example	Agent *i*, believing that agent *j* thinks that a shark is a mammal, attempts to change *j*'s belief.
	
	`(disconfirm`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((mammal shark))`
	`  :language FIPA-SL)`

3.7 Failure

Summary	The action of telling another agent that an action was attempted but the attempt failed.
Message Content	A tuple, consisting of an action expression and a proposition giving the reason for the failure.
Description	The *failure* act is an abbreviation for informing that an act was considered feasible by the sender, but was not completed for some given reason. The agent receiving a failure act is entitled to believe that: • the action has not been done, and, • the action is (or, at the time the agent attempted to perform the action, was) feasible The (causal) reason for the failure is represented by the proposition, which is the second element of the message content tuple. It may be the constant *true*. Often it is the case that there is little either agent can do to further the attempt to perform the action.
Formal Model	$\langle i,\ \text{failure}\ (j,\ a,\ \phi)\rangle \equiv$ $\langle i,\ \text{inform}\ (j,\ (\exists e)\ \text{Single}\ (e) \wedge \text{Done}\ (e,\ \text{Feasible}\ (a) \wedge$ $\text{I}_i\ \text{Done}\ (a)) \wedge \phi \wedge \neg\text{Done}\ (a) \wedge \neg\text{I}_i\ \text{Done}\ (a))\rangle$ FP: $\text{B}_i\ \alpha \wedge \neg\text{B}_i\ (\text{Bif}_j\ \alpha \vee \text{Uif}_j\ \alpha)$ RE: $\text{B}_j\ \alpha$ Where: $\alpha = (\exists e)\ \text{Single}\ (e) \wedge \text{Done}\ (e,\ \text{Feasible}\ (a) \wedge \text{I}_i\ \text{Done}\ (a)) \wedge \phi \wedge$ $\neg\text{Done}\ (a) \wedge \neg\text{I}_i\ \text{Done}\ (a)$ Agent *i* informs agent *j* that, in the past, *i* had the intention to do action *a* and *a* was feasible. *i* performed the action of attempting to do *a* (that is, the action/event *e* is the attempt to do *a*), but now *a* has not been done and *i* no longer has the intention to do *a*, and ϕ is true. The informal implication is that ϕ is the reason that the action failed, though this causality is not expressed formally in the semantic model.

Example	Agent *j* informs *i* that it has failed to open a file.
	
	`(failure`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (open "foo.txt"))`
	`  (error-message "No such file: foo.txt"))`
	`  :language FIPA-SL)`

3.8 Inform

Summary	The sender informs the receiver that a given proposition is true.
Message Content	A proposition.
Description	The sending agent: • holds that some proposition is true, • intends that the receiving agent also comes to believe that the proposition is true, and, • does not already believe that the receiver has any knowledge of the truth of the proposition. The first two properties defined above are straightforward: the sending agent is sincere, and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last property is concerned with the semantic soundness of the act. If an agent knows already that some state of the world holds (that the receiver knows proposition *p*), it cannot rationally adopt an intention to bring about that state of the world (*i.e.* that the receiver *comes to know p* as a result of the inform act). Note that the property is not as strong as it perhaps appears. The sender *is not* required to *establish* whether the receiver knows p. It is only the case that, in the case that the sender already happens to know about the state of the receiver's beliefs, it should not adopt an intention to tell the receiver something it already knows. From the receiver's viewpoint, receiving an inform message entitles it to believe that: • the sender believes the proposition that is the content of the message, and, • the sender wishes the receiver to believe that proposition also. Whether or not the receiver does, indeed, adopt belief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.
Formal Model	$\langle i,$ inform $(j, \phi)\rangle$ FP: $B_i\phi \wedge \neg B_i(\text{Bif}_j\phi \vee \text{Uif}_j\phi)$ RE: $B_j\phi$

Examples	Agent *i* informs agent *j* that (it is true that) it is raining today.
	`(inform`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  "weather (today, raining)"`
	`  :language Prolog)`

3.9 Inform If

Summary	A macro action for the agent of the action to inform the recipient whether or not a proposition is true.
Message Content	A proposition.
Description	The *inform-if* macro act is an abbreviation for informing whether or not a given proposition is believed. The agent which enacts an *inform-if* macro-act will actually perform a standard *inform* act. The content of the inform act will depend on the informing agent's beliefs. To *inform-if* on some closed proposition ϕ: • if the agent believes the proposition, it will inform the other agent that ϕ, and, • if it believes the negation of the proposition, it informs that ϕ is false, that is, $\neg\phi$. Under other circumstances, it may not be possible for the agent to perform this plan. For example, if it has no knowledge of ϕ, or will not permit the other party to know (that it believes) ϕ, it will send a *refuse* message.
Formal Model	$\langle i,$ inform-if $(j, \phi)\rangle \equiv$ $\langle i,$ inform $(j, \phi)\rangle \mid \langle i,$ inform $(j, \neg\phi)\rangle$ FP: $\mathrm{Bif}_i\ \phi \wedge \neg \mathrm{B}_i\ (\mathrm{Bif}_j\ \phi \vee \mathrm{Uif}_j\ \phi)$ RE: $\mathrm{Bif}_j\ \phi$ *Inform-if* represents two possible courses of action: *i* informs *j* that ϕ, or *i* informs *j* that not ϕ.

Examples	Agent *i* requests *j* to inform it whether Lannion is in Normandy.
	
	`(request`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (inform-if`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  "in( lannion, normandy)"`
	`  :language Prolog)))`
	`  :language FIPA-SL)`
	
	Agent *j* replies that it is not:
	
	`(inform`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  "\+ in (lannion, normandy)"`
	`  :language Prolog)`

3.10 Inform Ref

Summary	A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name.
Message Content	An object description (a referential expression).
Description	The *inform-ref* macro action allows the sender to inform the receiver some object that the sender believes corresponds to a descriptor, such as a name or other identifying description. *inform-ref* is a macro action, since it corresponds to a (possibly infinite) disjunction of *inform* acts, each of which informs the receiver that "the object corresponding to *name* is *x*" for some given x. For example, an agent can plan an *inform-ref* of the current time to agent *j*, and then perform the act "*inform j* that the time is 10.45". The agent performing the act should believe that the object or set of objects corresponding to the reference expression is the one supplied, and should not believe that the receiver of the act already knows which object or set of objects corresponds to the reference expression. The agent may elect to send a *refuse* message if it is unable to establish the preconditions of the act.
Formal Model	$\langle i, \text{inform-ref}\ (j, \mathit{Ref}\ x\ \delta(x))\rangle \equiv$ $\langle i, \text{inform}\ (j, \mathit{Ref}\ x\ \delta(x) = r_1)\rangle \mid \ldots \mid$ $(\langle i, \text{inform}\ (j, \mathit{Ref}\ x\ \delta(x) = r_k)\rangle$ FP: $\text{Bref}_i\ \mathit{Ref}\ x\ \delta(x) \wedge \neg B_i(\text{Bref}_j\ \mathit{Ref}\ x\ \delta(x) \vee \text{Uref}_j\ \mathit{Ref}\ x\ \delta(x))$ RE: $\text{Bref}_j\ \mathit{Ref}\ x\ \delta(x)$ Note: $\mathit{Ref}\ x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, $\text{any}\ x\ \delta(x)$ or $\text{all}\ x\ \delta(x)$. *Inform-ref* represents an unbounded, possibly infinite set of possible courses of action, in which *i* informs *j* of the referent of *x*.

Example	Agent *i* requests *j* to tell it the current Prime Minister of the United Kingdom:
	
	`(request`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (inform-ref`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  "((iota ?x (UKPrimeMinister ?x)))"`
	`  :ontology world-politics`
	`  :language FIPA-SL)))`
	`  :reply-with query0`
	`  :language FIPA-SL)`
	
	Agent *j* replies:
	
	`(inform`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((= (iota ?x (UKPrimeMinister ?x)) "Tony Blair"))`
	`  :ontology world-politics`
	`  :in-reply-to query0)`

	Note that a standard abbreviation for the *request* of *inform-ref* used in this example is the act *query-ref*.

3.11 Not Understood

Summary	The sender of the act (for example, *i*) informs the receiver (for example, *j*) that it perceived that *j* performed some action, but that *i* did not understand what *j* just did. A particular common case is that *i* tells *j* that *i* did not understand the message that *j* has just sent to *i*.
Message Content	A tuple consisting of an action or event, for example, a communicative act, and an explanatory reason.
Description	The sender received a communicative act that it did not understand. There may be several reasons for this: the agent may not have been designed to process a certain act or class of acts, or it may have been expecting a different message. For example, it may have been strictly following a pre-defined protocol, in which the possible message sequences are predetermined. The *not-understood* message indicates to that the sender of the original, that is, misunderstood, action that nothing has been done as a result of the message. This act may also be used in the general case for *i* to inform *j* that it has not understood *j*'s action. The second element of the message content tuple is a proposition representing the reason for the failure to understand. There is no guarantee that the reason is represented in a way that the receiving agent will understand. However, a co-operative agent will attempt to explain the misunderstanding constructively. Note: It is not possible to fully capture the intended semantics of an action not being understood by another agent. The characterization below captures that an event happened and that the recipient of the not-understood message was the agent of that event. ϕ must be a well formed formula of the content language of the sender agent. If the sender uses the bare textual message, that is, 'String' in the syntax definition, as the reason ϕ, it must be a propositional assertive statement and (at least) the sender can understand that (natural language) message and calculate its truth value, that is, decide its assertion is true or false. So, for example, in the SL language, to use textual message for the convenience of humans, it must be encapsulated as the constant argument of a predicate defined in the ontology that the sender uses, for example: `(error "message")`
Formal Model	$\langle i,$ not-understood$(j, a, \phi)\rangle \equiv$ $\langle i,$ inform$(j, \alpha)\rangle$ FP: $B_i\,\alpha \wedge \neg B_i\,(Bif_j\,\alpha \vee Uif_j\,\alpha)$ RE: $B_j\,\alpha$ Where: $\alpha = \phi \wedge (\exists x)\ B_i\ ((\iota e$ Done $(e) \wedge$ Agent $(e, j) \wedge B_j$(Done $(e) \wedge$ Agent $(e, j) \wedge (a = e))) = x)$

Examples	Agent *i* did not understand a *query-if* message because it did not recognize the ontology.
	
	`(not-understood`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`    ((action (agent-identifier :name j)`
	`       (query-if`
	`         :sender (agent-identifier :name j)`
	`         :receiver (set (agent-identifier :name i))`
	`         :content`
	`           "<`*`fipa-ccl content expression`*`>"`
	`         :ontology www`
	`         :language FIPA-CCL))`
	`     (unknown (ontology "www")))`
	`  :language FIPA-SL)`

3.12 Propagate

Summary	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received *propagate* message to them.
Message Content	A tuple of a descriptor, that is, a referential expression, denoting an agent or agents to be forwarded the *propagate* message, an embedded ACL communicative act, that is, an ACL message, performed by the sender to the receiver of the *propagate* message and a constraint condition for propagation, for example, timeout.
Description	This is a compound action of the following two actions. First, the sending agent requests the recipient to treat the embedded message in the received *propagate* message as if it is directly sent from the sender, that is, as if the sender performed the embedded communicative act directly to the receiver. Second, the sender wants the receiver to identify agents denoted by the given descriptor and to send a modified version of the received *propagate* message to them, as described below. On forwarding, the `:receiver` parameter of the forwarded *propagate* message is set to the denoted agent(s) and the `:sender` parameter is set to the receiver of the received *propagate* message. The sender and receiver of the embedded communicative act of the forwarded *propagate* message is also set to the same agent as the forwarded *propagate* message's sender and receiver, respectively. This communicative act is designed for delivering messages through federated agents by creating a chain (or tree) of *propagate* messages. An example of this is instantaneous brokerage requests using a *proxy* message, or persistent requests by a *request-when*/*request-whenever* message embedding a *proxy* message.
Formal Model	$\langle i,$ propagate $(j,$ *Ref* $x\ \delta(x),\ \langle i,$ *cact*$\rangle,\ \phi)\rangle \equiv$ $\langle i,$ *cact*$(j)\rangle$; $\langle i,$ inform $(j,\ I_i((\exists y)\ (B_j$ (*Ref* $x\ \delta(x) = y) \wedge$ Done $(\langle j,$ propagate $(y,$ *Ref* $x\ \delta(x),\ \langle j,$ *cact*$\rangle,\ \phi)\rangle,\ B_j\ \phi))))\rangle$ FP: FP (*cact*) $\wedge\ B_i\ \alpha \wedge \neg B_i$ (Bif$_j\ \alpha \vee$ Uif$_j\ \alpha$) RE: Done (*cact*) $\wedge\ B_j\ \alpha$ Where : $\alpha =$ $I_i((\exists y)\ (B_j$ (*Ref* $x\ \delta(x) = y) \wedge$ Done $(\langle j,$ propagate $(y,$ *Ref* $x\ \delta(x),\ \langle j,$ *cact*$\rangle,\ \phi)\rangle,\ B_j\ \phi)))$ Agent *i* performs the embedded *communicative act* to *j*: $\langle i,$ *cact*$(j)\rangle$ and *i* wants *j* to send the *propagate* message to the denoted agent(s) by *Ref* $x\ \delta(x)$. Note that $\langle i,$*cact*$\rangle$ in the propagate message is the ACL communicative act without the `:receiver` parameter. Note: *Ref* $x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$.

Example

Agent *i* requests agent *j* and its federating other brokerage agents to do brokering video-on-demand server agent to get "SF" programs.

```
(propagate
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :neme j))
  :content
    ((any ?x (registered
      (agent-description
        :name ?x
        :services (set
          (service-description
            :name agent-brokerage))))
     (action (agent-identifier :name i)
     (proxy
       :sender (agent-identifier :name i)
       :receiver (set (agent-identifier :name j))
       :content
         "((all ?y (registered
           (agent-description
             :name ?y
             :services (set
               (service-description
                 :name video-on-demand)))))
          (action (agent-identifier :name j)
          (request
            :sender (agent-identifier :name j)
            :content
              \"((action ?z[4]
                (send-program (category "SF"))))\"
            :ontology vod-server-ontology
            :protocol fipa-reqest …))
          true)"
       :ontology brokerage-agent-ontology
       :conversation-id vod-brokering-2
       :protocol fipa-brokering …))
     (< (hop-count) 5))
  :ontology brokerage-agent-ontology
  …)
```

[4] We cannot specify the concrete actor name when agent *i* sends the *propagate* message because it is identified by the referential expression `(all ?y …)`. In the above example, a free variable `?z` is used as the mandatory actor agent part of the action expression `send-program` in the content of embedded *request* message.

3.13 Propose

Summary	The action of submitting a proposal to perform a certain action, given certain preconditions.
Message Content	A tuple containing an action description, representing the action that the sender is proposing to perform, and a proposition representing the preconditions on the performance of the action.
Description	*Propose* is a general-purpose action to make a proposal or respond to an existing proposal during a negotiation process by proposing to perform a given action subject to certain conditions being true. The actual protocol under which the negotiation process is being conducted is known either by prior agreement, or is explicitly stated in the `:protocol` parameter of the message. The proposer (the sender of the *propose*) informs the receiver that the proposer will adopt the intention to perform the action once the given precondition is met, and the receiver notifies the proposer of the receiver's intention that the proposer performs the action. A typical use of the condition attached to the proposal is to specify the price of a bid in an auctioning or negotiation protocol.
Formal Model	$\langle i, \text{propose } (j, \langle i, act\rangle, \phi)\rangle \equiv$ $\langle i, \text{inform } (j, I_j \text{ Done } (\langle i, act\rangle, \phi) \Rightarrow I_i \text{ Done } (\langle i, act\rangle, \phi))\rangle$ FP: $B_i \alpha \wedge \neg B_i (Bif_j \alpha \vee Uif_j \alpha)$ RE: $B_j \alpha$ Where: $\alpha = I_j \text{ Done } (\langle i, act\rangle, \phi) \Rightarrow I_i \text{ Done } (\langle i, act\rangle, \phi)$ Agent *i* informs *j* that, once *j* informs *i* that *j* has adopted the intention for *i* to perform action *act*, and the preconditions for *i* performing *act* have been established, *i* will adopt the intention to perform *act*.

Example	Agent *j* proposes to *i* to sell 50 boxes of plums for $5. This example continues the example of *cfp*.
	
	`(propose`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`    ((action j (sell plum 50))`
	`     (= (any ?x (and (= (price plum) ?x) (< ?x 10))) 5)`
	`  :ontology fruit-market`
	`  :in-reply-to proposal2`
	`  :language FIPA-SL)`

3.14 Proxy

Summary	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.
Message Content	A tuple of a descriptor, that is, a referential expression, that denotes the target agents, an ACL communicative act, that is, an ACL message, to be performed to the agents, and a constraint condition for performing the embedded communicative act, for example, the maximum number of agents to be forwarded, etc.
Description	The sending agent informs the recipient that the sender wants the receiver to identify agents that satisfy the given descriptor, and to perform the embedded communicative act to them, that is, the receiver sends the embedded message to them. On performing the embedded communicative act, the `:receiver` parameter is set to the denoted agent and the `:sender` is set to the receiver of the *proxy* message. If the embedded communicative act contains a `:reply-to` parameter (for example, in the *recruiting* case where the `:protocol` parameter is set to *fipa-recruiting*), it should be preserved in the performed message. In the case of a *brokering* request (that is, the `:protocol` parameter is set to *fipa-brokering*), the brokerage agent (the receiver of the *proxy* message) must record some parameters, for example, `:conversation-id`, `:reply-with`, `:sender`, etc.) of the received *proxy* message to forward back the reply message(s) from the target agents to the corresponding requester agent (the sender of the *proxy* message).
Formal Model	$\langle i,$ proxy $(j,$ *Ref* $x\ \delta(x)), \langle j,$ *cact*$\rangle, \phi\rangle \equiv$ $\langle i,$ inform $(j, I_i((\exists y)(B_j$ (*Ref* $x\ \delta(x) = y) \wedge$ Done $(\langle j,$ *cact*$(y)\rangle, B_j\ \phi))))\rangle$ FP: $B_i\ \alpha \wedge \neg B_i$ (Bif$_j\ \alpha \vee$ Uif$_j\ \alpha$) RE: $B_j\ \alpha$ Where: $\alpha =$ $I_i((\exists y)$ $(B_j$ (*Ref* $x\ \delta(x) = y) \wedge$ Done $(\langle j,$ *cact*$(y)\rangle, B_j\ \phi)))$ Agent *i* wants *j* to perform the embedded communicative act to the denoted agent(s) (y) by *Ref* $x\ \delta(x)$. Note that $\langle j,$*cact*$\rangle$ in the proxy message is the ACL communicative act without the `:receiver` parameter. Its receiver is denoted by the given *Ref* $x\ \delta(x)$ by the agent *j*. Note: *Ref* $x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$. Two types of proxy can be distinguished. We will call the type of proxy defined above *strong*, because it is a feasibility precondition of *j*'s communicative act to *y* that *j* satisfies the feasibility preconditions of the proxied communicative act. So, if *i* proxies an *inform* of the proposition ψ to *y* via *j*, *j* must believe ψ before it sends the proxied inform message to *y*. In addition, we could define *weak-proxy,* where we do not suppose that *j* is required to believe ψ. In this case, *j* cannot directly inform *y* of ψ, because *j* does not satisfy the feasibility preconditions of *inform*. In this case, *j* can only *inform y* that the original sender *i* has the intention that the *inform* of ψ should happen. More generally, *weak-proxy* can be expressed as an instance of proxy where the action $\langle$j,*cact*(y)$\rangle$ is replaced by $\langle$j, inform(y, I_i Done $(\langle i,$ cact(y)$\rangle))\rangle$.

Example

Agent *i* requests agent *j* to do recruiting and request a video-on-demand server to send "SF" programs.

```
(proxy
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    ((all ?x (registered(agent-description
      :name ?x
      :services (set
        (service-description
          :name video-on-demand)))))
     (action (agent-identifier :name j)
       (request
         :sender (agent-identifier :name j)
         :content
           "((action  ?y[5]
              (send-program (category "SF"))))"
         :ontology vod-server-ontology
         :language FIPA-SL
         :protocol fipa-request
         :reply-to (set (agent-identifier :name i))
         :conversation-id request-vod-1)
      true)
  :language FIPA-SL
  :ontology brokerage-agent
  :protocol fipa-recruiting
  :conversation-id vod-brokering-1 …)
```

[5] We cannot specify the concrete actor name when agent *i* sends the *proxy* message because it is identified by the referential expression `(all ?x …)`. In the above example, a free variable `?x` is used as the mandatory actor agent part of the action expression `send-program` in the content of embedded *request* message.

3.15 Query If

Summary	The action of asking another agent whether or not a given proposition is true.
Message Content	A proposition.
Description	*Query-if* is the act of asking another agent whether (it believes that) a given proposition is true. The sending agent is requesting the receiver to *inform* it of the truth of the proposition. The agent performing the *query-if* act: • has no knowledge of the truth value of the proposition, and, • believes that the other agent can inform the querying agent if it knows the truth of the proposition.
Formal Model	$\langle i$, query-if (j, ϕ)$\rangle \equiv$ $\langle i$, request (j, $\langle j$, inform-if (i, ϕ)$\rangle$)$\rangle$ FP: $\neg Bif_i\phi \wedge \neg Uif_i\phi \wedge \neg B_i\ I_j$ Done($\langle$j, inform-if (i, ϕ)$\rangle$) RE: Done ($\langle$j, inform(i, ϕ)$\rangle$|$\langle$j, inform (i, $\neg\phi$)$\rangle$)

Example	Agent *i* asks agent *j* if *j* is registered with domain server d1:
	
	`(query-if`
	`:sender (agent-identifier :name i)`
	`:receiver (set (agent-identitfier :name j))`
	`:content`
	`((registered (server d1) (agent j)))`
	`:reply-with r09`
	`…)`
	
	Agent *j* replies that it is not:
	
	`(inform`
	`:sender (agent-identifier :name j)`
	`:receiver (set (agent-identifier :name i))`
	`:content ((not (registered (server d1) (agent j))))`
	`:in-reply-to r09)`

3.16 Query Ref

Summary	The action of asking another agent for the object referred to by an referential expression.
Message Content	A descriptor (a referential expression).
Description	*Query-ref* is the act of asking another agent to inform the requester of the object identified by a descriptor. The sending agent is requesting the receiver to perform an *inform* act, containing the object that corresponds to the descriptor. The agent performing the *query-ref* act: • does not know which object or set of objects corresponds to the descriptor, and, • believes that the other agent can inform the querying agent the object or set of objects that correspond to the descriptor.
Formal Model	$\langle i,$ query-ref $(j,$ *Ref* $x\ \delta(x))\rangle \equiv$ $\langle i,$ request $(j, \langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle)\rangle$ FP: $\neg \text{Bref}_i($*Ref* $x\ \delta(x)) \wedge \neg \text{Uref}_i($*Ref* $x\ \delta(x)) \wedge$ $\neg \text{B}_i\ \text{I}_j$ *Done*$(\langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle)$ RE: Done$(\langle i,$ inform $(j,$ *Ref* $x\ \delta(x) = r_1)\rangle$ |...| $\langle i,$ inform $(j,$ *Ref* $x\ \delta(x) = r_k)\rangle)$ Note: *Ref* $x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$.

Example	Agent *i* asks agent *j* for its available services.
	
	`(query-ref`
	`  :sender (agent-identinfier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((all ?x (available-service j ?x)))`
	`  …)`
	
	Agent *j* replies that it can reserve trains, planes and automobiles.
	
	`(inform`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((= (all ?x (available-service j ?x))`
	`  (set (reserve-ticket train)`
	`  (reserve-ticket plane)`
	`  (reserve automobile))))`
	`  …)`

3.17 Refuse

Summary	The action of refusing to perform a given action, and explaining the reason for the refusal.
Message Content	A tuple, consisting of an action expression and a proposition giving the reason for the refusal.
Description	The *refuse* act is an abbreviation for denying (strictly speaking, *disconfirm*ing) that an act is possible for the agent to perform, and stating the reason why that is so. The *refuse* act is performed when the agent cannot meet all of the preconditions for the action to be carried out, both implicit and explicit. For example, the agent may not know something it is being asked for, or another agent requested an action for which it has insufficient privilege. The agent receiving a *refuse* act is entitled to believe that: • the action has not been done, • the action is not feasible (from the point of view of the sender of the refusal), and, • the (causal) reason for the refusal is represented by the a proposition which is the second element of the message content tuple, (which may be the constant true). There is no guarantee that the reason is represented in a way that the receiving agent will understand. However, a cooperative agent will attempt to explain the refusal constructively. See the description at *not-understood*.
Formal Model	$\langle i, \text{refuse } (j, \langle i, act\rangle, \phi)\rangle \equiv$ $\langle i, \text{disconfirm } (j, \text{Feasible}(\langle i, act\rangle))\rangle;$ $\langle i, \text{inform } (j, \phi \wedge \neg\text{Done } (\langle i, act\rangle) \wedge \neg I_i \text{ Done } (\langle i, act\rangle))\rangle$ FP: $B_i \neg\text{Feasible } (\langle i, act\rangle) \wedge B_i (B_j \text{ Feasible } (\langle i, act\rangle) \vee U_j \text{ Feasible } (\langle i, act\rangle)) \wedge B_i \alpha \wedge \neg B_i (\text{Bif}_j \alpha \vee \text{Uif}_j \alpha)$ RE: $B_j \neg\text{Feasible } (\langle i, act\rangle) \wedge B_j \alpha$ Where: $\alpha = \phi \wedge \neg\text{Done } (\langle i, act\rangle) \wedge \neg I_i \text{ Done } (\langle i, act\rangle)$ Agent *i* informs *j* that action *act* is not feasible, and further that, because of proposition ϕ, *act* has not been done and *i* has no intention to do *act*.

Example	Agent *j* refuses to *i* reserve a ticket for *i*, since there are insufficient funds in *i*'s account.
	
	`(refuse`
	`  :sender (agent-identifier :name j)`
	`  :receiver (set (agent-identifier :name i))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (reserve-ticket LHR MUC 27-sept-97))`
	`  (insufficient-funds ac12345))`
	`  :language FIPA-SL)`

3.18 Reject Proposal

Summary	The action of rejecting a proposal to perform some action during a negotiation.
Message Content	A tuple consisting of an action description and a proposition which formed the original proposal being rejected, and a further proposition which denotes the reason for the rejection.
Description	*Reject-proposal* is a general-purpose rejection to a previously submitted proposal. The agent sending the rejection informs the receiver that it has no intention that the recipient performs the given action under the given preconditions. The additional proposition represents a reason that the proposal was rejected. Since it is in general hard to relate cause to effect, the formal model below only notes that the reason proposition was believed true by the sender at the time of the rejection. Syntactically the reason should be treated as a causal explanation for the rejection, even though this is not established by the formal semantics.
Formal Model	$\langle i,$ reject-proposal $(j, \langle j, act\rangle, \phi, \psi)\rangle \equiv$ $\langle i,$ inform $(j, \neg I_i$ Done $(\langle j, act\rangle, \phi) \wedge \psi)\rangle$ FP : $B_i\ \alpha \wedge \neg B_i$ (Bif$_j\ \alpha \vee$ Uif$_j\ \alpha$) RE : $B_j\ \alpha$ Where: $\alpha = \neg I_i$ Done($\langle j, act\rangle, \phi) \wedge \psi$ Agent *i* informs *j* that, because of proposition ψ, *i* does not have the intention for *j* to perform action *act* with precondition ϕ.

Example	Agent *i* informs *j* that it rejects an offer from *j* to sell.
	
	`(reject-proposal`
	`  :sender (agent-identifier :name i)`
	`  :receiver (set (agent-identifier :name j))`
	`  :content`
	`  ((action (agent-identifier :name j)`
	`  (sell plum 50))`
	`  (cost 200)`
	`  (price-too-high 50))`
	`  :in-reply-to proposal13)`

3.19 Request

Summary	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
Message Content	An action expression.
Description	The sender is requesting the receiver to perform some action. The content of the message is a description of the action to be performed, in some language the receiver understands. The action can be any action the receiver is capable of performing: pick up a box, book a plane flight, change a password, etc. An important use of the request act is to build composite conversations between agents, where the actions that are the object of the request act are themselves communicative acts such as *inform*.
Formal Model	$\langle i,$ request $(j, a)\rangle$ FP: FP (a) $[i\backslash j] \wedge B_i$ Agent $(j, a) \wedge \neg B_i\ I_j$ Done (a) RE: Done (a) FP(a) $[i\backslash j]$ denotes the part of the FPs of *a* which are mental attitudes of *i*.
Examples	Agent *i* requests *j* to open a file. (request :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content "open \"db.txt\" for input" :language vb)

3.20 Request When

Summary	The sender wants the receiver to perform some action when some given proposition becomes true.
Message Content	A tuple of an action description and a proposition.
Description	*Request-when* allows an agent to inform another agent that a certain action should be performed as soon as a given precondition, expressed as a proposition, becomes true. The agent receiving a *request-when* should either refuse to take on the commitment, or should arrange to ensure that the action will be performed when the condition becomes true. This commitment will persist until such time as it is discharged by the condition becoming true, the requesting agent *cancel*s the *request-when*, or the agent decides that it can no longer honour the commitment, in which case it should send a *refuse* message to the originator. No specific commitment is implied by the specification as to how frequently the proposition is re-evaluated, nor what the lag will be between the proposition becoming true and the action being enacted. Agents that require such specific commitments should negotiate their own agreements prior to submitting the *request-when* act.
Formal Model	$\langle i,$ request-when $(j, \langle j, act\rangle, \phi)\rangle \equiv$ $\langle i,$ inform $(j, (\exists e')$ Done $(e') \wedge$ Unique $(e') \wedge$ I_i Done $(\langle j, act\rangle, (\exists e)$ Enables $(e, B_j \phi) \wedge$ Has-never-held-since $(e', B_j \phi)))\rangle$ FP: $B_i \alpha \wedge \neg B_i$ (Bif$_j \alpha \vee$ Uif$_j \alpha$ RE: $B_j \alpha$ Where: $\alpha = (\exists e')$ Done (e') (Unique $(e') \wedge$ I_i Done $(\langle j, act\rangle, (\exists e)$ Enables $(e, B_j \phi) \wedge$ Has-never-held-since $(e', B_j \phi))$ Agent *i* informs *j* that *i* intends for *j* to perform some *act* when *j* comes to believe ϕ.
Examples	Agent *i* tells agent *j* to notify it as soon as an alarm occurs. (request-when :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content ((action (agent-identifier :name j) (inform :sender (agent-identifier :name j) :receiver (set (agent-identifier :name i)) :content "((alarm \"something alarming!\"))")) (Done(alarm))) …)

3.21 Request Whenever

Summary	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.
Message Content	A tuple of an action description and a proposition.
Description	*Request-whenever* allows an agent to inform another agent that a certain action should be performed as soon as a given precondition, expressed as a proposition, becomes true, and that, furthermore, if the proposition should subsequently become false, the action will be repeated as soon as it once more becomes true. *Request-whenever* represents a persistent commitment to re-evaluate the given proposition and take action when its value changes. The originating agent may subsequently remove this commitment by performing the *cancel* action. No specific commitment is implied by the specification as to how frequently the proposition is re-evaluated, nor what the lag will be between the proposition becoming true and the action being enacted. Agents who require such specific commitments should negotiate their own agreements prior to submitting the *request-when* act.
Formal Model	$\langle i,$ request-whenever $(j, \langle j, act\rangle, \phi)\rangle \equiv$ $\langle i,$ inform $(j,$ I_i Done $(\langle j, act\rangle, (\exists e)$ Enables $(e,$ B_j $\phi)))\rangle$ FP: $B_i\ \alpha \wedge \neg B_i\ (Bif_j\ \alpha \vee Uif_j\ \alpha)$ RE: $B_j\ \alpha$ Where: α = I_i Done $(\langle j, act\rangle, (\exists e)$ Enables $(e,$ B_j $\phi))$ Agent *i* informs *j* that *i* intends that *j* will perform some *act* whenever some event causes *j* to believe ϕ.
Examples	Agent *i* tells agent *j* to notify it whenever the price of widgets rises from less than 50 to more than 50. (request-whenever :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content ((action (agent-identifier :name j) (inform-ref :sender (agent-identifier :name j) :receiver (set (agent-identifier :name i)) :content "((iota ?x (= (price widget) ?x)))")) (> (price widget) 50)) …)

3.22 Subscribe

Summary	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.
Message Content	A descriptor (a referential expression).
Description	The *subscribe* act is a persistent version of *query-ref*, such that the agent receiving the *subscribe* will *inform* the sender of the value of the reference, and will continue to send further *inform*s if the object denoted by the description changes. A subscription set up by a *subscribe* act is terminated by a *cancel* act.
Formal Model	$\langle i,$ subscribe $(j,$ *Ref* $x\ \delta(x))\rangle \equiv$ $\langle i,$ request-whenever $(j, \langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle,$ $(\exists y)\ B_j\ ((\textit{Ref}\ x\ \delta(x) = y))\rangle$ FP: $B_i\ \alpha \wedge \neg B_i\ (Bif_j\ \alpha \vee Uif_j\ \alpha)$ RE: $B_j\ \alpha$ Where: $\alpha =$ I_i Done $(\langle j,$ inform-ref $(i,$ *Ref* $x\ \delta(x))\rangle,$ $(\exists e)$ Enables $(e,\ (\exists y)\ B_j\ ((\textit{Ref}\ x\ \delta(x) = y)))$ Note: *Ref* $x\ \delta(x)$ is one of the referential expressions: $\iota x\ \delta(x)$, any $x\ \delta(x)$ or all $x\ \delta(x)$.
Examples	Agent *i* wishes to be updated on the exchange rate of Francs to Dollars, and makes a subscription agreement with *j* (an exchange rate server). (subscribe :sender (agent-identifier :name i) :receiver (set (agent-identifier :name j)) :content ((iota ?x (= ?x (xch-rate FFR USD)))))

4 References

[Cohen90] Cohen, P. R. and Levesque, H. J., Intention is Choice with Commitment. In: Artificial Intelligence, 42(2-3), pages 213-262, 1990.

[FIPA00008] FIPA SL Content Language Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00008/

[FIPA00025] FIPA Interaction Protocol Library Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00025/

[FIPA00070] FIPA ACL Message Representation in String. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00070/

[Garson84] Garson, G. W., Quantification in Modal Logic. In: Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic, Gabbay, D., & Guentner, F., editors. D. Reidel Publishing Company, pages 249-307, 1984.

[Halpern85] Halpern, J. Y. and Moses, Y., A Guide to the Modal Logics of Knowledge and Belief: A Preliminary Draft. In: Proceedings of the IJCAI-85, 1985.

[Sadek90] Sadek, M. D., Logical Task Modelling for Man-Machine Dialogue. In: Proceedings of AAAI90, pages 970-975, Boston, USA, 1990.

[Sadek91a] Sadek, M. D., Attitudes Mentales et Interaction Rationnelle: Vers une Théorie Formelle de la Communication. Thèse de Doctorat Informatique, Université de Rennes I, France, 1991.

[Sadek91b] Sadek, M. D., Dialogue Acts are Rational Plans. In: Proceedings of the ESCA/ETRW Workshop on the Structure of Multimodal Dialogue, pages 1-29, Maratea, Italy, 1991.

[Sadek92] Sadek, M. D., A Study in the Logic of Intention. In: Proceedings of the 3rd Conference on Principles of Knowledge Representation and Reasoning (KR92), pages 462-473, Cambridge, USA, 1992.

[Searle69] Searle, J.R., Speech Acts. Cambridge University Press, 1969.

5 Informative Annex A — Formal Basis of ACL Semantics

This section provides a formal definition of the communication language and its semantics. The intention here is to provide a clear, unambiguous reference point for the standardised meaning of the inter-agent communicative acts expressed through messages and protocols. This section of the specification is *normative*, in that agents which claim to conform to the FIPA specification ACL must behave in accordance with the definitions herein. However, this section may be treated as informative in the sense that no new information is introduced here that is not already expressed elsewhere in this document. The non mathematically-inclined reader may safely omit this section without sacrificing a full understanding of the specification.

Note also that *conformance testing*, that is, demonstrating in an unambiguous way that a given agent implementation is correct with respect to this formal model, is not a problem which has been solved in this FIPA specification. Conformance testing will be the subject of further work by FIPA.

5.1 Introduction to the Formal Model

This section presents, in an informal way, the model of communicative acts that underlies the semantics of the message language. This model is presented only in order to ground the stated meanings of communicative acts and protocols. It **is not a proposed architecture** or a structural model of the agent design.

Other than the special case of agents that operate singly and interact only with human users or other software interfaces, agents must communicate with each other to perform the tasks for which they are responsible. Consider the basic case shown in *Figure 1*.

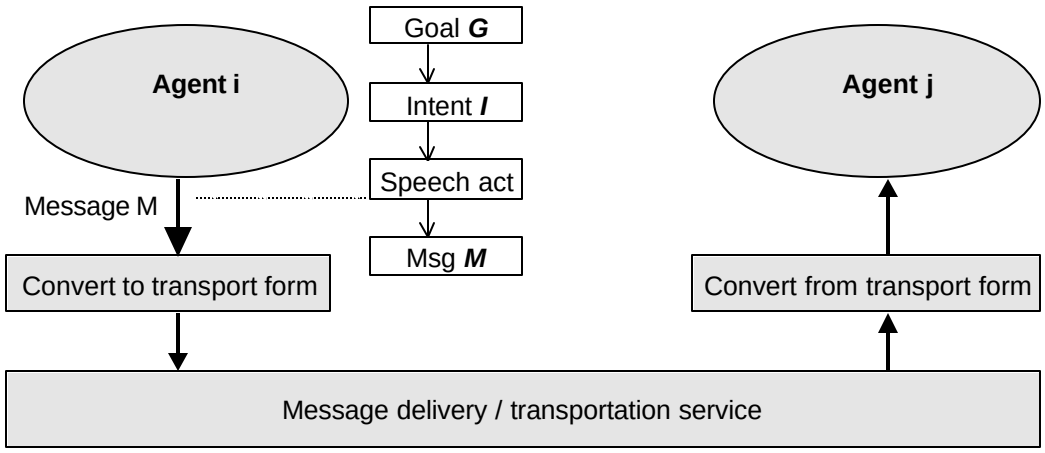


Figure 1: Message Passing Between Two Agents

Suppose that, in abstract terms, Agent i has amongst its *mental attitudes* the following: some goal or objective ***G*** and some intention ***I***. Deciding to satisfy ***G***, the agent adopts a specific intention, ***I***. Note that neither of these statements entail a commitment on the design of Agent i: ***G*** and ***I*** could equivalently be encoded as explicit terms in the mental structures of a BDI agent, or implicitly in the call stack and programming assumptions of a simple Java or database agent.

Assuming that Agent i cannot carry out the intention by itself, the question then becomes which message or set of messages should be sent to another agent (j in *Figure 1*) to assist or cause intention ***I*** to be satisfied? If Agent i is behaving in some reasonable sense rationally, it will not send out a message whose effect will not satisfy the intention and hence achieve the goal. For example, if Harry wishes to have a barbecue (***G*** = "have a barbecue"), and thus derives a goal to find out if the weather will be suitable (***G'*** = "know if it is raining today"), and thus intends to find out the weather (***I*** = "find out if it is raining"), he will be ill-advised to ask Sally "have you bought Acme stock today?" From Harry's perspective, whatever Sally says, it will not help him to determine whether it is raining today.

Continuing the example, if Harry, acting more rationally, asks Sally "can you tell me if it is raining today?", he has acted in a way he hopes will satisfy his intention and meet his goal (assuming that Harry thinks that Sally will know the answer). Harry can reason that the effect of asking Sally is that Sally would tell him, hence making the request fulfils his intention. Now, having asked the question, can Harry actually assume that, sooner or later, he will know whether it is raining? Harry *can* assume that Sally knows that he does not know, *and* that she knows that he is asking her to tell him. But, simply on the basis of having asked, Harry cannot assume that Sally will act to tell him the weather: she is independent, and may, for example, be busy elsewhere.

In summary: an agent plans, explicitly or implicitly (through the construction of its software) to meet its goals ultimately by communicating with other agents, that is, sending messages to them and receiving messages from them. The agent will select acts based on the relevance of the act's expected outcome or *rational effect* to its goals. However, it cannot assume that the rational effect will necessarily result from sending the messages.

5.2 The Semantic Language

The Semantic Language (SL[6]) is the formal language used to define the semantics of the FIPA ACL. As such, SL itself has to be precisely defined. In this section, we present the SL language definition and the semantics of the primitive communicative acts.

5.2.1 Basis of the Semantic Language Formalism

In SL, logical propositions are expressed in a logic of mental attitudes and actions, formalised in a first order modal language with identity[7] (see [Sadek 91a] for details of this logic). The components of the formalism used in the following are as follows:

- p, p_1, ... are taken to be closed formulas denoting propositions,
- ϕ and ψ are formula schemas, which stand for any closed proposition,
- i and j are schematic variables which denote agents, and,
- $\models \phi$ means that ϕ is valid.

The mental model of an agent is based on the representation of three primitive attitudes: *belief*, *uncertainty* and *choice* (or, to some extent, *goal*). They are respectively formalised by the modal operators B, U, and C. Formulas using these operators can be read as:

- $B_i p$ "i (implicitly) believes (that) p",
- $U_i p$ "i is uncertain about p but thinks that p is more likely than $\neg p$", and,
- $C_i p$ "i desires that p currently holds".

The logical model for the operator B is a *KD45* possible-worlds-semantics Kripke structure (see, for example, [Halpern85]) with the fixed domain principle (see, for example, [Garson84]).

To enable reasoning about action, the universe of discourse involves, in addition to individual objects and agents, sequences of events. A sequence may be formed with a single event. This event may be also the *void* event. The language involves terms (in particular a variable e) ranging over the set of event sequences.

To talk about complex *plans*, events (or actions) can be combined to form *action expressions*:

[6] SL is also used for the content language of the FIPA ACL messages (see [FIPA00008]).
[7] This logical framework is similar in many aspects to that of [Cohen90].

- a_1 ; a_2 is a *sequence* in which a_2 follows a_1

- a_1 | a_2 is a *nondeterministic choice*, in which either $a_1$happens or a_2, but not both.

Action expressions will be noted as *a*.

The operators *Feasible*, *Done* and *Agent* are introduced to enable reasoning about actions, as follows:

- Feasible (*a*, *p*) means that *a* can take place and if it does *p* will be true just after that,

- Done (*a*, *p*) means that *a* has just taken place and *p* was true just before that,

- Agent (*i*, *a*) means that *i* denotes the only agent that ever performs (in the past, present or future) the actions which appear in action expression *a*,

- Single *(a)* means that *a* denotes an action expression that is not a sequence. Any individual action is *Single*. The composite act *a* ; *b* is not *Single*. The composite act *a* | *b* is *Single* iff both a and b are *Single*.

From belief, choice and events, the concept of *persistent goal* is defined. An agent *i* has *p* as a persistent goal, if *i* has *p* as a goal and is self-committed toward this goal until *i* comes to believe that the goal is achieved or to believe that it is unachievable. *Intention* is defined as a persistent goal imposing the agent to act. Formulas as $PG_i p$ and $I_i P$ are intended to mean that "*i* has *p* as a persistent goal" and "*i* has the intention to bring about *p*", respectively. The definition of *I* entails that *intention generates a planning process*. See [Sadek92] for the details of a formal definition of intention.

Note that there is no restriction on the possibility of embedding mental attitude or action operators. For example, formula $U_i B_j I_j Done(a, B_i p)$ informally means that agent *i* believes that, probably, agent *j* thinks that *i* has the intention that action *a* be done before which *i* has to believe *p*.

A fundamental property of the proposed logic is that the modelled agents are perfectly in agreement with their own mental attitudes. Formally, the following schema is valid:

$\phi \Leftrightarrow B_i\phi$

where ϕ is governed by a modal operator formalising a mental attitude of agent *i*.

5.2.2 Abbreviations

In the text below, the following abbreviations are used:

1. Feasible (*a*) ≡ Feasible (*a*, True)

2. Done (*a*) ≡ Done (*a*, True)

3. Possible (ϕ) ≡ ($\exists a$) Feasible (*a*, ϕ)

4. $Bif_i\phi \ \circ B_i\phi \vee B_i\neg\phi$
   $Bif_i\phi$ means that either agent *i* believes ϕ or that it believes $\neg\phi$.

5. $Bref_i\ \iota x\delta(x) \circ (\exists y)B_i(\iota x\delta(x) = y)$
   where ι is the operator for definite description and $\iota x\delta(x)$ is read "the (*x* which is) δ". $Bref_i\ \iota x\delta(x)$ means that agent *i* believes that it knows the (*x* which is) δ.

6. $Uif_i\phi \equiv U_i\phi \vee U_i\neg\phi$
   $Uif_i\phi$ means that either agent *i* is uncertain (in the sense defined above) about ϕ or that it is uncertain about $\neg\phi$.

7. $Uref_i\ \iota x\delta(x) \equiv (\exists y)U_i(\iota x\delta(x) = y)$

Uref$_i$ $\iota x\delta(x)$ has the same meaning as *Bref*$_i$ $\iota x\delta(x)$, except that agent *i* has an uncertainty attitude with respect to $\delta(x)$ instead of a belief attitude.

8. $AB_{n,i,j}\phi \equiv B_iB_jB_i \ldots \phi$
introduces the concept of *alternate beliefs*, *n* is a positive integer representing the number of *B* operators alternating between *i* and *j*.

In the text, the term "knowledge" is used as an abbreviation for "believes or is uncertain of".

5.3 Underlying Semantic Model

The components of a communicative act (CA) model that are involved in a planning process characterise both the reasons for which the act is selected and the conditions that have to be satisfied for the act to be planned. For a given act, the former is referred to as the *rational effect* or *RE*[8], and the latter as the *feasibility preconditions* or *FP*s, which are the qualifications of the act.

5.3.1 Property 1

To give an agent the capability of planning an act whenever the agent intends to achieve its RE, the agent should adhere to the following property:

Let a_k be an act such that:

1. $(\exists x)\ B_i a_k = x$

2. *p* is the RE of a_k and

3. $\neg C_i \neg$Possible (Done(a_k));

then the following formula is valid:

$I_i p \Rightarrow I_i$ Done ($a_1 | \ldots | a_n$)

Where:

$a_1, \ldots, a_n$ are *all* the acts of type a_k.

This property says that an agent's intention to achieve a given goal generates an intention that one of the acts known to the agent be done. Further, the act is such that its rational effect corresponds to the agent's goal, and that the agent has no reason for not doing it.

The set of feasibility preconditions for a CA can be split into two subsets: the *ability preconditions* and the *context-relevance preconditions*. The ability preconditions characterise the intrinsic ability of an agent to perform a given CA. For instance, to *sincerely assert* some proposition *p*, an agent has to believe that *p*. The context-relevance preconditions characterise the relevance of the act to the context in which it is performed. For instance, an agent can be intrinsically able to make a promise while believing that the promised action is not needed by the addressee. The context-relevance preconditions correspond to the Gricean quantity and relation maxims.

5.3.2 Property 2

This property imposes on an agent an intention to seek the satisfiability of its FPs, whenever the agent elects to perform an act by virtue of property 1[9]:

[8] Rational effect is also referred to as the *perlocutionary effect* in some of the work prior to this specification (see [Sadek90]).
[9] See [Sadek91b] for a generalised version of this property.

$\models I_i \text{Done}(a) \Rightarrow B_i \text{Feasible}(a) \vee I_i B_i \text{Feasible}(a)$

5.3.3 Property 3

If an agent has the intention that (the illocutionary component of) a communicative act be performed, it necessarily has the intention to bring about the rational effect of the act. The following property formalises this idea:

$\models I_i \text{ Done } (a) \Rightarrow I_i \text{ RE } (a)$

Where:

RE (*a*) denotes the rational effect of act *a*.

5.3.4 Property 4

Consider now the complementary aspect of CA planning: the consuming of CAs. When an agent observes a CA, it should believe that the agent performing the act has the intention (to make public its intention) to achieve the rational effect of the act. This is called the *intentional effect*. The following property captures this intuition:

$\models B_i(\text{Done } (a) \wedge \text{Agent } (j, a) \Rightarrow I_j \text{RE } (a))$

Note, for completeness only, that a strictly precise version of this property is as follows:

$\models B_i(\text{Done } (a) \wedge \text{Agent } (j, a) \Rightarrow I_j \, B_i \, I_j \text{ RE } (a))$

5.3.5 Property 5

Some FPs persist after the corresponding act has been performed. For the particular case of CAs, the next property is valid for all the FPs which do not refer to time. In such cases, when an agent observes a given CA, it is entitled to believe that the persistent feasibility preconditions hold:

$\models B_i(\text{Done (a)} \Rightarrow \text{FP (a)})$

5.3.6 Notation

A communicative act model will be presented as follows:

<*i*, *act (j*, *C)*>

FP: ϕ_1

RE: ϕ_2

where *i* is the agent of the act, *j* the recipient, *act* the name of the act, *C* stands for the semantic content or propositional content[10], and ϕ_1 and ϕ_2 are propositions. This notational form is used for brevity, only within this section on the formal basis of ACL. The correspondence to the standard transport syntax (see [FIPA00070]) adopted above is illustrated by a simple translation of the above example:

```
(act
  :sender i
  :receiver j
  :content
    C)
```

[10] See [Searle69] for the notions of *propositional content* (and *illocutionary force*) of an *illocutionary act*.

Note that this also illustrates that some aspects of the operational use of the FIPA ACL fall outside the scope of this formal semantics but are still part of the specification. For example, the above example is actually incomplete without `:language` and `:ontology` parameters to given meaning to *C*, or some means of arranging for these to be known.

5.3.7 Note on the Use of Symbols in Formulae

Note that variable symbols are used in the semantics description formulae of each communicative act as shown in *Table 1*.

Symbol	Usage
a	Used to denote an action. Example: $a = \langle i, \text{inform}\ (j, p)\rangle$
act	Used to denote an action type. Example: $act = \text{inform}\ (j, p)$ Thus, if $a = \langle i, \text{inform}\ (j, p)\rangle$ and $act = \text{inform}\ (j, p)$ then $a = \langle i, act\rangle$.
cact	Used to denace only an ACL communicative act type.
ϕ	Used to denote any closed proposition (without any restriction).
p	Used to denote a given proposition. Thus 'φ' is a formula schema, that is, a variable that denotes a formula, and 'p' is a formula (not a variable).

Table 1: Meaning of Symbols in Formulae

Consider the following axiom examples:

$I_i\ \phi \Rightarrow \neg B_i\ \phi$,

Here, ϕ stands for any formula. It is a variable.

$B_i\ (\text{Feasible}\ (a) \Leftrightarrow p)$

Here, *p* stands for a given formula: the FP of act *'a'*.

5.3.8 Supporting Definitions

$\text{Enables}\ (e, \phi) = \text{Done}\ (e, \neg\phi) \wedge \phi$

$\text{Has-never-held-since}\ (e', \phi) = (\forall e_1)\ (\forall e_2)\ \text{Done}\ (e'; e_1\ ;\ e_2) \Rightarrow \text{Done}\ (e_2, \neg\phi)$

5.4 Primitive Communicative Acts

5.4.1 The Assertive Inform

One of the most interesting assertives regarding the core of mental attitudes it encapsulates is the act of *informing*. An agent *i* is able to *inform* an agent *j* that some proposition *p* is true *only* if *i* believes *p* (that is, only if $B_i p$). This act is considered to be context-relevant only if *i* does not think that *j* already believes *p* or its negation, or that *j* is uncertain about *p* (recall that belief and uncertainty are mutually exclusive). If *i* is already aware that *j* does already believe *p*, there is no need for further action by *i*. If *i* believes that *j* believes *not p*, *i* should *disconfirm p*. If *j* is uncertain about *p*, *i* should *confirm p*.

$\langle i, \text{INFORM}\ (j, \phi)\rangle$
FP: $B_i\phi \wedge \neg B_i(\text{Bif}_j\phi \vee \text{Uif}_j\phi)$
RE: $B_j\phi$

The FPs for *inform* have been constructed to ensure mutual exclusiveness between CAs, when more that one CA might deliver the same rational effect.

Note, for completeness only, that the above version of the *inform* model is the operationalised version. The complete theoretical version (regarding the FPs) is the following:

$\langle i, \text{INFORM } (j, \phi)\rangle$
FP: $B_i\phi \wedge \bigwedge_{n>1} \neg AB_{n,i,j} \neg B_i\phi \wedge \neg B_iB_j\phi \wedge \bigwedge_{n>2} \neg AB_{n,i,j} B_j\phi$
RE: $B_j\phi$

5.4.2 The Directive Request

The following model defines the directive *request*:

$\langle i, \text{REQUEST } (j, a)\rangle$
FP: FP (a) $[i\backslash j] \wedge B_i$ Agent $(j, a) \wedge B_i \neg PG_j$ Done (a)
RE: Done (a)

Where:

- *a* is a schematic variable for which any action expression can be substituted,
- *FP (a)* denotes the feasibility preconditions of *a*, and,
- *FP (a) [i\j]* denotes the part of the FPs of *a* which are mental attitudes of *i*.

5.4.3 Confirming an Uncertain Proposition: Confirm

The rational effect of the act *confirm* is identical to that of most of the assertives, *i*.e., the addressee comes to believe the semantic content of the act. An agent *i* is able to *confirm* a property *p* to an agent *j only* if *i* believes *p* (that is, B_ip). This is the sincerity condition an assertive act imposes on the agent performing the act. The act *confirm* is context-relevant *only* if *i* believes that *j* is uncertain about *p* (that is, $B_i U_j p$). In addition, the analysis to determine the qualifications required for an agent to be entitled to perform an *Inform* act remains valid for the case of the act *confirm*. These qualifications are identical to those of an *inform* act for the part concerning the ability preconditions, but they are different for the part concerning the context relevance preconditions. Indeed, an act *confirm* is irrelevant if the agent performing it believes that the addressee is not uncertain of the proposition intended to be *confirm*ed.

In view of this analysis, the following is the model for the act *confirm:*

$\langle i, \text{CONFIRM } (j, \phi)\rangle$
FP: $B_i\phi \wedge B_iU_j\phi$
RE: $B_j\phi$

5.4.4 Contradicting Knowledge: Disconfirm

The *confirm* act has a negative counterpart: the *disconfirm* act. The characterisation of this act is similar to that of the *confirm* act and leads to the following model:

$\langle i, \text{DISCONFIRM } (j, \phi)\rangle$
FP: $B_i\neg\phi \wedge B_i(U_j\phi \vee B_j\phi)$
RE: $B_j\neg\phi$

5.5 Composite Communicative Acts

An important distinction is made between acts that can be carried out directly, and those macro acts which can be planned (which includes requesting another agent to perform the act), but cannot be directly carried out. The distinction centres on whether it is possible to say that an act has been done, formally *Done (Action, p)*. An act which is composed of primitive communicative actions (inform, request, confirm), or which is composed from primitive messages by substitution or sequencing (via the ";" operator), can be performed directly and can be said afterwards to be done. For example, agent *i* can inform *j* that *p*; *Done (<i, inform(j, p)>)* is then true, and the meaning (that is, the rational effect) of this action can be precisely stated.

However, a large class of other useful acts is defined by composition using the disjunction operator (written "|"). By the meaning of the operator, only one of the disjunctive components of the act will be performed when the act is carried out. A good example of these macro-acts is the *inform-ref* act. *Inform-ref* is a macro act defined formally by:

$$\langle i, \text{INFORM-REF } (j, \iota x\ \delta(x)) \rangle \equiv$$
$$\langle i, \text{INFORM } (j, \iota x\ \delta(x) = r_1) \rangle \mid \ldots \mid \langle i, \text{INFORM } (j, \iota x\ \delta(x) = r_n) \rangle$$

where n may be infinite. This act may be requested (for example, j may request i to perform it), or i may plan to perform the act in order to achieve the (rational) effect of j knowing the referent of $\delta(x)$. However, when the act is actually performed, what is sent, and what can be said to be *Done*, is an *inform* act.

Finally an inter-agent plan is a sequence of such communicative acts, using either composition operator, involving two or more agents. FIPA interaction protocols (see [FIPA00025]) are primary examples of pre-enumerated inter-agent plans.

5.5.1 The Closed Question Case

In terms of illocutionary acts, exactly what an agent *i* is *requesting* when uttering a sentence such as "Is *p*?" toward a recipient *j*, is that *j* performs the act of "*informing i that p*" or that *j* performs the act "*informing i that* $\neg p$". We know the model for both of these acts: $\langle j, \textit{INFORM } (i, \phi) \rangle$. In addition, we know the relation "*or*" that holds between these two acts: it is the relation that allows for the building of action expressions which represent a *non-deterministic choice* between several (sequences of) events or actions.

In fact, as mentioned above, the semantic content of a directive refers to an *action expression*; so, this can be a *disjunction* between two or more acts. Hence, by using the utterance "Is *p*?", what an agent *i requests* an agent *j* to do is the following action expression:

$$\langle j, \text{INFORM } (i, p) \rangle \mid \langle j, \text{INFORM } (i, \neg p) \rangle$$

It seems clear that the semantic content of a directive realised by a yes/no-question can be viewed as an action expression characterising an indefinite choice between two CAs *inform*. In fact, it can also be shown that the binary character of this relation is only a special case: in general, any number of CAs *inform* can be handled. In this case, the addressee of a directive is allowed to choose one among several acts. This is not only a theoretical generalisation: it accounts for classical linguistic behaviour traditionally called *alternatives question*. An example of an utterance realising an alternative question is "Would you like to travel in first class, in business class, or in economy class?" In this case, the semantic content of the *request* realised by this utterance is the following action expression:

$$\langle j, \text{INFORM } (i, p_1) \rangle \mid \langle j, \text{INFORM } (i, p_2) \rangle \mid \langle j, \text{INFORM } (i, p_3) \rangle$$

Where p_1, p_2 and p_3 are intended to mean respectively that *j* wants to travel in first class, in business class, or in economy class.

As it stands, the agent designer has to provide the plan-oriented model for this type of action expression. In fact, it would be interesting to have a model which is not specific to the action expressions characterising the non-deterministic choice between CAs of type *inform*, but a more general model where the actions referred to in the disjunctive relation remain unspecified. In other words, to describe the preconditions and effects of the expression $a_1 \mid a_2 \mid \ldots \mid a_n$ where $a_1, a_2, \ldots, a_n$ are any action expressions. It is worth mentioning that the goal is to characterise this action expression as a *disjunctive*

macro-act which is planned as such; we are not attempting to characterise the non-deterministic choice between acts which are planned separately. In both cases, the result is a branching plan but in the first case, the plan is branching in an *a priori* way while in the second case it is branching in an *a posteriori* way.

An agent will plan a macro-act of non-deterministic choice when it intends to achieve the rational effect of one of the acts composing the choice, *no matter which one it is*. To do that, one of the feasibility preconditions of the acts must be satisfied, *no matter which one it is*. This produces the following model for a disjunctive macro-act:

$a_1 \mid a_2 \mid \ldots \mid a_n$
  FP: $FP\ (a_1) \vee FP\ (a_2) \vee \ldots \vee FP\ (a_n)$
  RE: $RE\ (a_1) \vee RE\ (a_2) \vee \ldots \vee RE\ (a_n)$

Where $FP\ (a_k)$ and $RE\ (a_k)$ represent the FPs and the RE of the action expression a_k, respectively.

Because the yes/no-question, as shown, is a particular case of alternatives question, the above model can be specialised to the case of two acts *inform* having opposite semantic contents. Thus, we get the following model:

$\langle i, \text{INFORM}\ (j, \phi)\rangle \mid \langle i, \text{INFORM}\ (j, \neg\phi)\rangle$
  FP: $\text{Bif}_i\phi \wedge \neg\text{B}_{i(}\text{Bif}_j\phi \vee \text{Uif}_j\phi)$
  RE: $\text{Bif}_j\phi$

In the same way, we can derive the disjunctive macro-act model which gathers the acts *confirm* and *disconfirm*. We will use the abbreviation $\langle i, CONFDISCONF\ (j, \phi)\rangle$ to refer to the following model:

$\langle i, \text{CONFIRM}\ (j, \phi)\rangle\ \phi\ \langle i, \text{DISCONFIRM}\ (j, \phi)\rangle)$
  FP: $\text{Bif}_i\phi \wedge \text{B}_i\text{U}_j\phi$
  RE: $\text{Bif}_j\phi$

5.5.2 The Query If Act

Starting from the act models $\langle j, INFORM\text{-}IF\ (i, \phi)\rangle$ and $\langle i, REQUEST\ (j, a)\rangle$, it is possible to derive the query-if act model (and not plan, as shown below). Unlike a confirm/disconfirm-question, which will be addressed below, a query-if act requires the agent performing it not to have any knowledge about the proposition whose truth value is asked for. To get this model, a transformation[11] has to be applied to the FPs of the act $\langle j, INFORM\text{-}IF\ (i, \phi)\rangle$ and leads to the following model for a *query-if* act:

$\langle i, \text{QUERY-IF}\ (j, \phi)\rangle \equiv$
  $\langle i, \text{REQUEST}\ (j, \langle j, \text{INFORM-IF}\ (i, \phi)\rangle\ )\rangle$
  FP: $\neg\text{Bif}_i\phi \wedge \neg\text{Uif}_i\phi \wedge \text{B}_i\ \neg\text{PG}_j\ \text{Done}\ (\langle j, \text{INFORM-IF}\ (i, \phi)\rangle)$
  RE: $\text{Done}(\ \langle j, \text{INFORM}\ (i, \phi)\rangle \mid \langle j, \text{INFORM}\ (i, \neg\phi)\rangle)$

5.5.3 The Confirm/Disconfirm Question Act

In the same way, it is possible to derive the following *confirm*/*disconfirm* question act model:

$\langle i, \text{REQUEST}\ (j, \langle j, \text{CONFDISCONF}\ (\ i, \phi)\rangle)\rangle$
  FP: $\text{U}_i\phi \wedge \text{B}_i\neg\text{PG}_j Done\ (\langle j, \text{CONFDISCONF}\ (i, \phi)\rangle)$
  RE: $Done\ (\langle j, \text{CONFIRM}\ (i, \phi)\rangle \mid \langle j, \text{DISCONFIRM}\ (i, \phi)\ \phi)$

[11] For more details about this transformation, called the *double-mirror transformation*, see [Sadek91a] and [Sadek91b].

5.5.4 The Open Question Case

Open question is a question which does not suggest a choice and, in particular, which does not require a yes/no answer. A particular case of open questions are the questions which require referring expressions as an answer. They are generally called *wh-questions*. The "wh" refers to interrogative pronouns such as "what", "who", "where", or "when". Nevertheless, this must not be taken literally since the utterance "How did you travel?" can be considered as a wh-question.

A formal plan-oriented model for the wh-questions is required. In the model below, *from the addressee's viewpoint*, this type of question can be viewed as a closed question where the suggested choice is not made explicit because it is *too wide*. Indeed, a question such as "What is your destination?" can be restated as 'What is your destination: Paris, Rome,... ?".

The problem is that, in general, the set of definite descriptions among which the addressee can (and must) choose is potentially an infinite set, not because, referring to the example above, there may be an infinite number of destinations, but because, theoretically, each destination can be referred to in potentially an infinite number of ways. For instance, Paris can be referred to as "the capital of France", "the city where the Eiffel Tower is located", "the capital of the country where the Man-Rights Chart was founded", *etc*. However, it must be noted that in the context of man-machine communication, the language used is finite and hence the number of descriptions acceptable as an answer to a *wh-question* is also finite.

When asking a *wh-question*, an agent *j* intends to acquire from the addressee *i* an identifying referring expression (IRE) [Sadek90] for a definite description, in the general case. Therefore, agent *j* intends to make his interlocutor *i* perform a CA which is of the following form:

$$\langle i, \text{INFORM}\ (j, \iota x \delta(x) = r) \rangle$$

Where *r* is an IRE, for example, a standard name or a definite description, and $\iota x \delta(x)$ is a definite description. Thus, the semantic content of the directive performed by a wh-question is a disjunctive macro-act composed with acts of the form of the act above. Here is the model of such a macro-act:

$$\langle i, \text{INFORM}(\ j, \iota x \delta(x) = r_1\ ) \rangle \mid \ldots \mid \langle i, \textit{INFORM}(\ j, \iota x \delta(x) = r_k\ ) \rangle$$

Where r_k are IREs. To deal with the case of closed questions, the generic plan-oriented model proposed for a disjunctive macro-act can be instantiated for the account of the macro-act above. Note that the following equivalence is valid:

$$(\text{B}_i\ \iota x \delta(x) = r_1 \vee \text{B}_i\ \iota x \delta(x) = r_2 \vee \ldots\ ) \Leftrightarrow (\exists y)\ \text{B}_i\ \iota x \delta(x) = y$$

This produces the following model, which is referred to as $\langle i, \textit{INFORM-REF}(\ j, \iota x\ \delta(x)\ ) \rangle$:

$\langle i, \text{INFORM-REF}(\ j, \iota x\ \delta(x)\ ) \rangle$
FP: $\text{Bref}_i\ \iota x\ \delta(x) \wedge \neg\ \text{B}_i\ (\text{Bref}_j\ \iota x\ \delta(x) \vee \text{Uref}_j\ \iota x\ \delta(x))$
RE: $\text{Bref}_j\ \iota x\ \delta(x)$

Where $\textit{Bref}_j\ \iota x \delta(x)$ and $\textit{Uref}_j\ \iota x \delta(x)$ are abbreviations introduced above, and $\textit{aref}_j\ \iota x \delta(x)$ is an abbreviation defined as:

$$\alpha\text{ref}_j\ \iota x\ \delta(x) \equiv \text{Bref}_j\ \iota x ? \delta(x) \vee \text{Uref}_j\ \iota x ? \delta(x)$$

Provided the act models $\langle j, \textit{INFORM-REF}\ (i, \iota x\ \delta(x)) \rangle$ and $\langle i, \textit{REQUEST}\ (j, a) \rangle$, *the wh-question* act model can be built up in the same way as for the yn-question act model. Applying the same transformation to the FPs of the act schema $\langle j, \textit{INFORM-REF}\ (i, \iota x \delta(x)) \rangle$, and by virtue of property 3, the following model is derived:

$\langle i, \text{QUERY-REF}\ (\ j, \phi) \rangle ? \equiv \langle i, \text{REQUEST}\ (j, \langle j, \text{INFORM-REF}\ (i, \iota x\ \delta(x) \rangle) \rangle$
FP: $\neg\alpha\text{ref}_i\ \iota x \delta(x) \wedge \text{B}_i\ \neg\text{PG}_j\ \text{Done}\ (\langle j, \text{INFORM-REF}\ (i, \iota x \delta(x)) \rangle)$
RE: $\text{Done}\ (\langle j, \text{INFORM}\ (i, \iota x \delta(x) = r_1\ ) \rangle \mid \ldots \mid \langle j, \text{INFORM}\ (i, \iota x \delta(x) = r_k\ ) \rangle)$

5.6 Inter-Agent Communication Plans

The properties of rational behaviour stated above in the definitions of the concepts of rational effect and of feasibility preconditions for CAs suggest an algorithm for CA planning. A plan is built up by this algorithm builds up through the inference of causal chain of intentions, resulting from the application of properties 1 and 2.

With this method, it can be shown that what are usually called "*dialogue acts*" and for which models are postulated, are, in fact, complex plans of interaction. These plans can be derived from primitive acts, by using the principles of rational behaviour. The following is an example of how such plans are derived.

The interaction plan "hidden" behind a question act can be more or less complex depending on the agent mental state when the plan is generated.

Let a *direct question* be a question underlain by a plan which is limited to the reaction strictly legitimised by the question. Suppose that the main content of *i*'s mental state is:

$B_i\ Bif_j\ \phi,\ I_i\ Bif_i\ \phi$

By virtue of property 1, the intention is generated that the act <*j*, *INFORM-IF* (*i*, ϕ)> be performed. Then, according to property 2, there follows the intention to bring about the feasibility of this act. Then, the problem is to know whether the following belief can be derived at that time from *i*'s mental state:

$B_i(Bif_j\ \phi \wedge (\neg B_j\ Bif_i\ \phi \vee Uif_i\ \phi))$

This is the case with *i*'s mental state. By virtue of properties 1 and 2, the intention that the act <*i*, *REQUEST* (*j*, <*j*, *INFORM-IF* (*i*, ϕ)>)> be done and then the intention to achieve its feasibility, are inferred. The following belief is derivable:

$B_i(\neg Bif_i\ \phi \wedge \neg Uif_i\ \phi)$

Now, no intention can be inferred. This terminates the planning process. The performance of a direct strict-yn-question plan can be started by uttering a sentence such as "Has the flight from Paris arrived?", for example.

Given the FPs and the RE of the plan above, the following model for a *direct strict-yn-question plan* can be established:

<*i*, YNQUESTION (*j*, ϕ)>
  FP: $B_i\ Bif_j\ \phi \wedge \neg Bif_i\ \phi \wedge \neg Uif_i\ \phi \wedge B_i\ \neg B_j(\ Bif_i\ \phi \vee Uif_i\ \phi)$
  RE: $Bif_i\ \phi$

XML and Agent Communication

XML provides a way of structuring data in hunam readable form. Part fo the meaning of data comes from its structure but XML alone cannot capture all the meaning by itself. You still need semantics provided by, for example, speech act theory, and SL.

XML lends itself well to expressing ontologies. The DTD allows people to define the structural relationships anong agreed upon terms. The resulting XML documents are then readable by humans and usable, without modification, by machines. Becaue of this convenience, a XML is increasing used to write standards, such as agent communiction standards.

Both FIPA and JADE have integrated some XML into their work. Here are some examples.

Exampels of XML in Agent Communication

ACL in XML

There is now an experimental DTD for ACL. You can look at the current proposed standard. At FIPA. Or Locally.

Here is the DTD from that document. You can see the influence of speech act theory, just as in the case of KQML.

<!-- Document Type: XML DTD
Document Purpose: Encoding of FIPA ACL messages in XML
(see [FIPA00067]) and http://www.fipa.org/)
Last Revised: 2000/03/07
-->

<!-- Possible FIPA Communicative Acts. See [FIPA00037] for a
full list of valid performatives.
-->
<!ENTITY % communicative-acts
"accept-proposal|agree|cancel|cfp|confirm
|disconfirm|failure|inform|not-understood
|propose|query-if|query-ref|refuse
|reject-proposal|request|request-when
|request-whenever|subscribe|inform-if
|inform-ref|proxy|propagate">

```
<!-- The FIPA message root element, the communicative act is
an attribute - see below and the message itself is a list
of parameters. The list is unordered. None of the elements
should occur more than once except receiver.
-->
<!ENTITY %msg-param
"receiver|sender|content|language|content-language-encoding|ontology|
protocol|reply-with|in-reply-to|reply-by|reply-to|conversation-id">

<!ELEMENT fipa-message (%msg-param;)*>

<!-- Attribute for the fipa-message - the communicative act itself and
the conversation id (which is here so an ID value can be used).
-->
<!ATTLIST fipa-message act (%communicative-acts;) #REQUIRED
conversation-id ID #IMPLIED>

<!-- The agent identifier of the sender.
-->
<!ELEMENT sender (agent-identifier)>

<!-- The agent identifier(s) of the receiver.
-->
<!ELEMENT receiver (agent-identifier)>

<!-- The message content.
One can choose to embed the actual content in the message,
or alternatively refer to a URI which represents this content
-->
<!ELEMENT content (#PCDATA)>
<!ATTLIST content href CDATA #IMPLIED>

<!-- The content language used for the content.
The linking attribute href associated with language can be used
to refer in an unambiguous way to the (formal) definition of the
standard/fipa content language.
-->
<!ELEMENT language (#PCDATA)>
```

<!ATTLIST language href CDATA #IMPLIED>

<!-- The encoding used for the content language.
The linking attribute href associated with encoding can be used
to refer in an unambiguous way to the (formal) definition of the
language encoding.
-->
<!ELEMENT content-language-encoding (#PCDATA)>
<!ATTLIST content-language-encoding href CDATA #IMPLIED>

<!-- The ontology used in the content.
The linking attribute href associated with ontology can be used
to refer in an unambiguous way to the (formal) definition of the
ontology.
-->
<!ELEMENT ontology (#PCDATA)>
<!ATTLIST ontology href CDATA #IMPLIED>

<!-- The protocol element.
The linking attribute href associated with protocol can be used
to refer in an unambiguous way to the (formal) definition of the
protocol.
-->
<!ELEMENT protocol (#PCDATA)>
<!ATTLIST protocol href CDATA #IMPLIED>

<!-- The reply-with parameter.
-->
<!ELEMENT reply-with (#PCDATA)>
<!ATTLIST reply-with href CDATA #IMPLIED>

<!-- The in-reply-to parameter.
-->
<!ELEMENT in-reply-to (#PCDATA)>
<!ATTLIST in-reply-to href CDATA #IMPLIED>

<!-- The reply-by parameter.
-->

```
<!ELEMENT reply-by EMPTY>

<!-- See [FIPA00071] for the definition of time.
-->
<!ATTLIST reply-by time CDATA #REQUIRED
href CDATA #IMPLIED>

<!-- The reply-to parameter.
-->
<!ELEMENT reply-to (agent-identifier)>

<!-- The conversation-id parameter.
-->
<!ELEMENT conversation-id (#PCDATA)>
<!ATTLIST conversation-id href CDATA #IMPLIED>

<!ELEMENT agent-identifier (name, addresses?, resolvers?, user-defined*)>

<!ELEMENT name EMPTY>

<!-- An id can be used to uniquely identify the name of the agent.
The refid attribute can be used to refer to an already defined
agent name, avoiding unnecessary repetition. Either the id
OR refid should be specified, (both should not be present at the
same time)
-->
<!ATTLIST name id ID #IMPLIED
refid IDREF #IMPLIED>

<!ELEMENT addresses (url+)>
<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA #IMPLIED>

<!ELEMENT resolvers (agent-identifier+)>

<!ELEMENT user-defined (#PCDATA)>
<!ATTLIST user-defined href CDATA #IMPLIED>
```

JADE has an add-on which provides a codec for the above DTD.

RDF in XML

One of the newest semantic languages used for agent communcation is the Resource Description Framework (RDF). FIPA sets the standard for this and the JADE team is promising to have an implementation by the end of 2001. RDF is a content language, a competitor for SL. RDF is supported by yet another important standards setting body, W3C, the World Wide Web Consortium.

RDF Specification HTML pdf (local)

[top] [previous] [next] Questions?

Agent Content Languages

SL

SL stands for Semantic Language. It is the standard language for use in the :content slot of ACL. Actually, as far as ACL communictive acts are concerned, any language can be used for content, e.g., prolog or english. However, most people doing agent research use some version of SL, SLO, SL1, or SL2.

To use SL you must adhere to the standards set by FIPA.

FIPA SL Specification

- pdf format (local)
- HTML format

SL is a subtle language and not easy to use. Not surprising since it is concerned with semantics, always a difficult topic. The JADE system uses it by default.

Other Content Languages

KIF

KIP stands for Knowledge Interchange Format. It was originally designed as a content language to work with KQML.

FIPA KIF specification

RDF

RDF stands for Resource Description Framework. It is quite new and has the advantage of being written in XML.

FIPA RDF specification

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA RDF Content Language Specification

Document title	FIPA RDF Content Language Specification		
Document number	XC00011A	**Document source**	FIPA TC C
Document status	Experimental	**Date of this status**	2000/08/18
Supersedes	FIPA00003		
Contact	fab@fipa.org		
Change history			
2000/08/18	Approved for Experimental		

Geneva, Switzerland

Foreword

The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. FIPA makes the results of its activities available to all interested parties and intends to contribute its results to the appropriate formal standards bodies.

The members of FIPA are individually and collectively committed to open competition in the development of agent-based applications, services and equipment. Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international organization without restriction. In particular, members are not bound to implement or use specific agent-based standards, recommendations and FIPA specifications by virtue of their participation in FIPA.

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Procedures for Technical Work. A complete overview of the FIPA specifications and their current status may be found in the FIPA List of Specifications. A list of terms and abbreviations used in the FIPA specifications may be found in the FIPA Glossary.

FIPA is a non-profit association registered in Geneva, Switzerland. As of January 2000, the 56 members of FIPA represented 17 countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found at http://www.fipa.org/.

Contents

1 Introduction

This specification describes how the Resource Description Framework (RDF - see [W3crdf]) can be used as content language in a FIPA message. Although FIPA does not require that a content language is able to represent actions[1], a lot of communicative acts require actions in their content. Therefore, we will show how RDF schemas can be defined extending its model to express:

- **Objects** which are constructs that represent an identifiable entity (be it abstract or concrete) in the domain of discourse,
- **Propositions** which are statements expressing that some sentence in a language is true or false, and,
- **Actions** which try to express an activity that can be carried out by an object.

By means of existing mechanisms in RDF (schema definitions), modular RDF extensions will be proposed, based on the `fipa-rdf0` content language. Those extensions will be able to tackle for example rules, logic algebra constructs, and others. These extensions can then be labelled as `fipa-rdf1`, `fipa-rdf2`, etc.

The general motivation behind this approach is to enable and ease the use of RDF Schemas emerging on the Web such as CC/PP, and to define one common standard approach here to increase the level of interoperability. The main strengths of the RDF language are in its extensibility, reusability and simplicity. Another advantage of RDF is that data and schemas are exchanged in a similar way.

The RDF model proposes the eXtensible Markup Language (XML - see [W3Cxml]) as an encoding syntax, but does not prevent anyone from using alternative encoding schemes. All `fipa-rdf` examples will therefore use an XML encoding, although, in principle, other encoding schemes could be used.

1.1 A Summary of RDF

The RDF framework is based on an entity-relationship model. The RDF Data Model is described by means of resources, properties and their values. A specific resource together with one or more named properties plus the values of these properties is an RDF description (a collection of RDF statements).

In addition to the RDF Data Model, the RDF Schemas (see [W3Crdfsch]) specification provides a typing system for the resources and properties used in the RDF data. It defines concepts such as classes, subclasses, properties or sub-properties. It also allows expressing constraints. Both the RDF Data Model and RDF Schema propose XML as a serialization syntax.

RDF is a "foundation for processing meta-data in the way that it provides interoperability between applications that exchange machine-understandable information." This suggests that RDF could be most useful to facilitate knowledge sharing and exchange between agents.

[1] A content language must be able to express at least any of propositions, objects or actions.

2 RDF as a FIPA Content Language

To be able to use RDF as a content language for FIPA ACL messages, we have to explore how objects, propositions and functions can be expressed in RDF, without endangering key extensibility inherent to the language. On the other hand, we will try to preserve RDF's simplicity, which is crucial for the success of languages on the Internet.

We suggest to use the name `fipa-rdf0`, for the combined use of RDF and the basic schemas which define the extensions needed for FIPA.

2.1 Objects

Taking the above into account, it is obvious to see an analogy between an ACL object and an RDF resource, since both are defined as descriptions of a certain identifiable entity. This enables us to use RDF resource identifiers and references as ACL object identifiers and references. This means that to resolve an RDF reference, we can use a the FIPA communicative act `query-ref` (see [FIPA00054]), which will then be followed by an 'inform' message, describing this object.

2.2 Propositions

In the same context it seems logical to model ACL propositions using RDF statements. An RDF statement is composed out of three parts: subject (resource), predicate (property) and object (literal/value). As an example, consider the sentence "W. Richard Stevens is the author of TCP/IP Illustrated". The RDF components of this proposition are the subject (TCP/IP Illustrated), the predicate (Author) and the object (W. Richard Stevens). This sentence/statement can then be described in RDF in the following manner:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:s="http://description.org/schema/">

  <rdf:Description ID="TCP/IP Illustrated">
    <s:author>W. Richard Stevens</s:author>
  </rdf:Description>
</rdf:RDF>
```

Figure 1 represents this in RDF graph form. This way we have a starting point to state logical expressions in our content. Taking this one step further, we can say that by expressing this statement, we indicate our belief in this statement. In this way we can say that we always assume that an RDF statement expresses a belief. This approach would be sufficient in any context where the level of logic involved is limited.

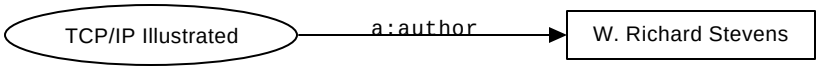


Figure 1: A Proposition as an RDF Statement

To overcome this shortcoming however, we will explain how logical belief or disbelief of a certain statement could be expressed explicitly using RDF. To express that we believe a statement to be true or false, we have to model the original statement as a reified statement, that is, a resource with four predefined properties:

- The **subject** property identifies the resource being described by the modelled statement; that is, the value of this property is the resource about which the original statement was made.

- The **predicate** property identifies the property of the original statement; that is, the value is the specific property in the original statement.

- The **object** property identifies the property value in the original statement; that is, the value is the object in the original statement.

- The value of the **type** property describes the type of the new resource. All reified statements are instances of `rdf:Statement`.

A new resource with the above four properties represents the original statement and can both be used as the object of another statement and have additional statements made about it. The resource with these four properties is not a replacement for the original statement, but it is a model of the statement.

By extending the RDF syntax model with the following elements, a means to express belief or disbelief of a statement is allowed (the complete schema of the RDF extensions can be found in *Section 4.1, RDF Schemas for FIPA RDF 1*):

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">

  <rdfs:Class rdf:ID="http://www.fipa.org/schemas#Proposition">
    <rdfs:label xml:lang="en">proposition</rdfs:label>
    <rdfs:label xml:lang="fr">proposition</rdfs:label>
    <rdfs:comment>This describes the set of propositions</rdfs:comment>
    <rdfs:subClassOf rdf:resource=
     "http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
  </rdfs:Class>

  <rdfs:ConstraintProperty rdf:ID="http://www.fipa.org/schemas#belief">
    <rdfs:label xml:lang="en">belief</rdfs:label>
    <rdfs:label xml:lang="fr">acte</rdfs:label>
    <rdfs:domain rdf:resource="#Proposition"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>
</rdf:RDF>
```

Using this method we can easily describe ACL propositions in RDF. As an example, the following proposition will be modelled: "The statement 'W. Richard Stevens is the author of TCP/IP Illustrated' is true". One way to do this is as follows:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:fipa=http://www.fipa.org/schemas/fipa-rdf0#">

  <fipa:Proposition>
    <rdf:subject>TCP/IP Illustrated</rdf:subject>
    <rdf:predicate rdf:resource="http://description.org/ schema#author"/>
    <rdf:object>W. Richard Stevens</rdf:object/>
    <fipa:belief>true</fipa:belief>
  </fipa:Proposition>
</rdf:RDF>
```

Expressing that the same statement is false, is equally easy by replacing the value 'true' with 'false'. The RDF graph representation of the 'false' statement is presented in *Figure 2*.

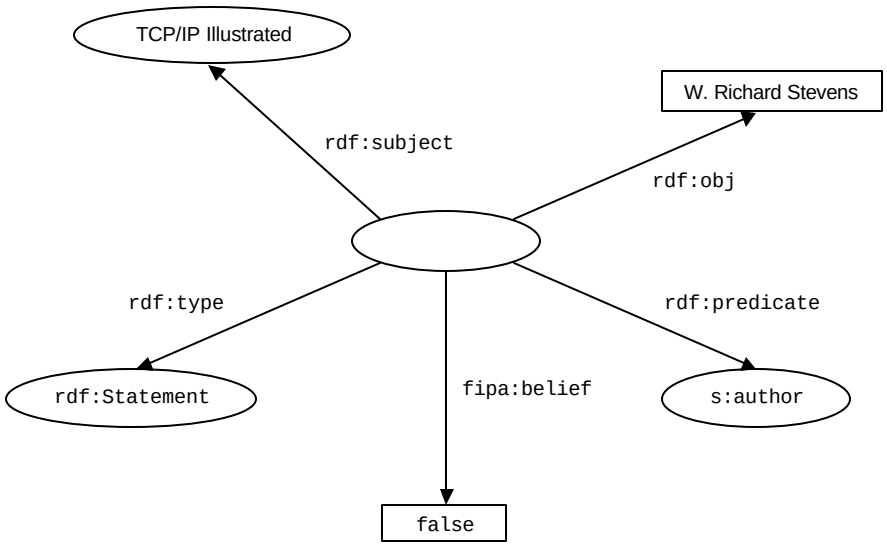


Figure 2: Explicit Logical Proposition in RDF

2.3 Actions

An **action** expresses an activity, carried out by an object. There are three different properties related to an 'action':

- An **act** identifies the operative part of the action; it can serve to identify the type of act or merely to describe the act. In the latter case specific types of action classes can be derived from the Action class.

- An **actor** identifies the entity responsible for the execution of the action, that is, the value is the specific entity which will/can/should perform the act (often the receiver, but possibly another agent/entity under "control" of the receiver).

- An **argument** identifies an (optional) entity which can be used for the execution of the action; that is, the value is entity which is used by the actor to perform the act. An action can have multiple arguments.

When looking at an action this way, there is a structural analogy with a RDF statement.

To model an action, the RDF syntax model can be extended with a new RDF type `fipa:Action` which has these properties. As an example, the following action will be modelled: "John opens door1 and door2". In this small example, the properties are the act (Open), the actor (John) and the arguments (door1 and door2). In RDF, this action can then be described as:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

  <fipa:Action rdf:ID="JohnAction1">
    <fipa:actor>John</fipa:actor>
    <fipa:act>open</fipa:act>
    <fipa:argument>
      <rdf:bag>
        <rdf:li>door1</rdf:li>
        <rdf:li>door2</rdf:li>
      </rdf:bag>
    </fipa:argument>
  </fipa:Action>
</rdf:RDF>
```

According to the RDF specification, the resource type defined in the schema corresponding to the type property can be used directly as an element name when the `Description` element contains a type property. So, a shorter version of the above example could be written as follows:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:fipa="http://www.fipa.org/schemas#">

  <fipa:Action rdf:ID="JohnAction1">
    <fipa:actor>John</fipa:actor>
    <fipa:act>open</fipa:act>
    <fipa:argument>
      <rdf:bag>
        <rdf:li>door1</rdf:li>
        <rdf:li>door2</rdf:li>
      </rdf:bag>
    </fipa:argument>
  </fipa:Action>
</rdf:RDF>
```

The model above still lacks the ability to state whether some action has finished or what the result is of the action. this can be solved by simply adding extra properties to the description of the action.

As an example, suppose Mary requests John to open door 1 and door 2 and then wants John to inform her if he performed the action and what the result is. This little scenario exists of two messages:

- Request from Mary to John containing the description of the action, and,
- Inform from John to Mary, referring to the action and stating the completion of the action.

Using FIPA ACL combined with RDF content, the first messages could be expressed as:

```
(request
  :sender Mary
  :receiver John
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas#">

      <fipa:Action rdf:ID="JohnAction1">
        <fipa:actor>John</rdf:actor>
        <fipa:act>open</rdf:act>
        <fipa:argument>
          <rdf:bag>
            <rdf:li>door1</rdf:li>
            <rdf:li>door2</rdf:li>
          </rdf:bag>
        </fipa:argument>
      </fipa:Action>
    </rdf:RDF>)
  :language fipa-rdf0)
```

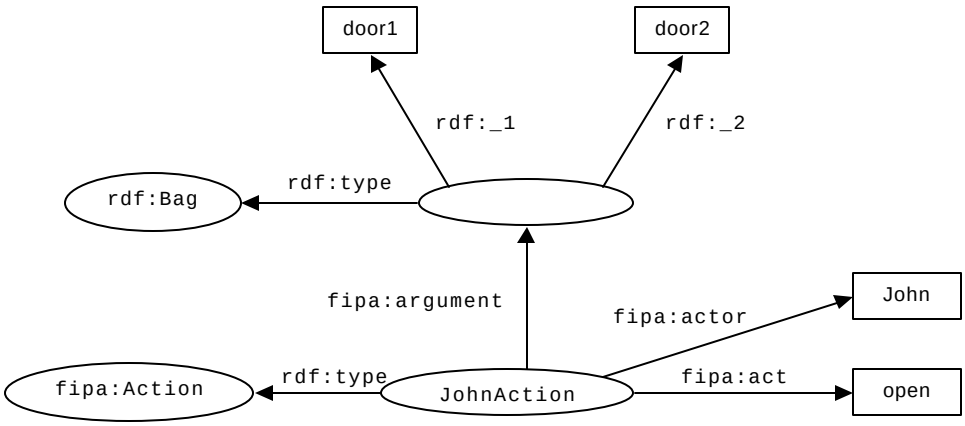


Figure 3: Example of an Open Action

And the subsequent reply message could be:

```
(inform
  :sender John
  :receiver Mary
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas#">

      <rdf:Description about="#JohnAction1">
        <fipa:done>true</fipa:done>
        <fipa:result>doors closed</fipa:result>
      </rdf:Description>
    </rdf:RDF> )
  :language fipa-rdf0)
```

Note the ability offered by RDF to include previous actions by means of a reference instead of repeating the whole action. The RDF graph representation of the complete action description is presented in *Figure 3*.

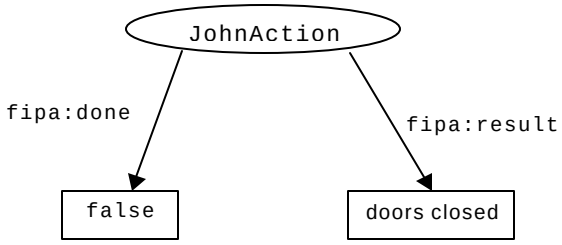


Figure 4: Result of an Open Action

2.4 Action Implementations

Different possible scenarios can be distinguished between when using the RDF actions. One possible usage is when a software designer describes in documentation (that is, in the RDF schemas in `rdfs:comment`) what is meant by a particular action; it is left to the implementer to decide which functions will be called. In another scenario, a more explicit description of the semantics might be needed by linking the action with some programming language. This section deals with the latter case.

When an agent does not know how to perform an action and needs a more explicit representation of this action, the sender agent can specify the code which implements the action. For this purpose a new property for actions is introduced, called `implementedBy`, which has a resource of the type `Code` as property its value.

A first possibility is that the property `implementedBy` contains a reference (a URI) to an external software module written in a specific programming language. For this purposes the `Code` resource therefore has a property `language` and a property `code-uri`. For reasons of simplicity, it is assumed that the language used is either Java or a scripting language such as JScript or ECMAscript. So, the property `code-uri` is a reference to the location of code where the method or function can be found (for Java a code base followed by a class name).

When a Java class is referenced, `code-uri` can contain the Java code-base. The receiving agent can then download this class, instantiate it (if needed), and perform the required action (or not). When a non-static class is being referred, we assume that there is always a zero-argument constructor (cfr. the requirement for JavaBeans).

In addition, we assume that there always exists a one-to-one correspondence between the FIPA arguments and fipa result property, into the method's arguments resp. return value. When multiple arguments are used, and the sequence of those is important, one should use the `rdf:Seq` container to separate them.

As an example, suppose agent 'Student' requests agent 'Mathematician' to find the next prime following after '7'. The request message is as follows (see *Figure 5*):

```
(request
  :sender Student
  :receiver Mathematician
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

      <fipa:Action rdf:ID="studreq01">
        <fipa:actor>Mathematician</fipa:actor>
        <fipa:act>findNextPrime</fipa:act>
        <fipa:argument>7</fipa:argument>
          <fipa:implementedBy>
            <fipa:Code>
            <fipa:language>Java</fipa:language>
              <fipa:code-uri>
                http://www.mathagent.com/math.utility.prime
              </fipa:code-uri>
            </fipa:Code>
          </fipa:implementedBy>
        </fipa:argument>
      </fipa:Action>
    </rdf:RDF> )
  :language fipa-rdf0)
```

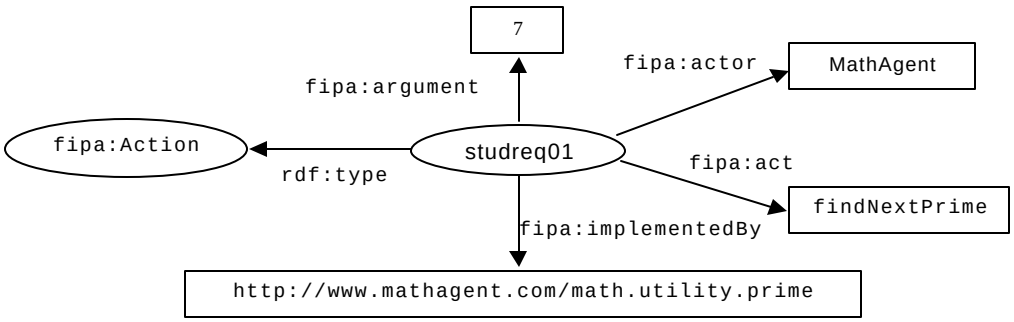


Figure 5: Actions and Implementation References

In the previous example, it is assumed that there exists a function or method in the static class `math.utility.prime.class` with the same name of the FIPA act (`findNextPrime`). If the name of the method is different from the FIPA act's name, then the method name should be included after the hash sign (#) of the property value `code-uri`. For example:

```
<fipa:implementedBy>
  <fipa:Code>
    <fipa:language>Java</fipa:language>
    <fipa:code-uri >
      http://www.mathagent.com/math.utility.prime#nextPrime
    </fipa:code-uri>
  </fipa:Code>
</fipa:implementedBy>
```

The Mathematician agent could reply with:

```
(inform
  :sender Mathematician
  :receiver Student
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

      <fipa:Action rdf:about="#studreq01">
        <fipa:done>true</fipa:done>
        <fipa:result>11</fipa:result>
      </fipa:Action>
    </rdf:RDF>)
  :language fipa-rdf0)
```

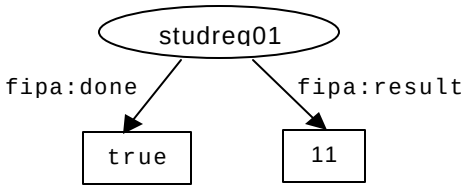


Figure 6: Result of the `findNextPrime` Action

Sometimes, multiple implementations can be associated with one specific action so the `implementedBy` property can contain an `rdf:Alt` container of Code classes. In some cases, the method implementation of the code may need to refer to values of the RDF data model and conventions are needed to establish a mapping between the RDF data and

(Java) object model. Although no real standards already exist, several initiatives are taking off to define such a binding. Examples include:

- DATAX: the Java interface (see [DATAX]),
- GINF: the interfaces specified in the Generic Interoperability Framework (see [Melnik99]),
- 3AP: the RDF-Java mapping as used in Alcatel's 3AP platform, and,
- Other Java API's have been suggested on the RDF-DEV mailing lists.

The following example shows the use of the binding property:

```
(request
  :sender agent-dealer
  :receiver agent-carshop
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

      <fipa:Action rdf:ID="price-update1">
        <fipa:actor>agent-carshop</rdf:actor>
        <fipa:act>addNewPrices</rdf:act>
        <fipa:implementedBy>
          <fipa:Code>
            <fipa:language>Java</fipa:language>
            <fipa:binding>DATAX</fipa:binding>
            <fipa:codeURI>
              http://www.carshop.com/bin/CarStock
            </fipa:codeURI>
          </fipa:Code>
        </fipa:implementedBy>
      </fipa:Action>
    </rdf:RDF> )
  :language fipa-rdf0)
```

The file `CarStock.java` could look as follows:

```
import com.muze.datax.*;
import com.muze.datax.rdf.*;

public class CarStock {

public CarStock() { }

void addNewPrices() {
  EntitySet entities = new RDFReader().read("carstock.rdf");
  DATAXFactory f = new DefaultDATAXFactory();
  Iterator it = entities.iterator();

  while (it.hasNext()) {
    Entity e = (Entity)it.next();
    Property p = e.getProperty("http://www.carshop.com/schemas#price");
    Float price = Float.valueOf(p.getValue());
    p-new = f.createProperty(Property.ATTRIBUTE,
```

```
            "newprice", 1.05*price.floatValue());
    e.add(p-new);
  }
}
```

In this example, the car dealer requests the car shop to attach new prices to their car stock: the new prices should become 5% higher than the old ones. In the Java file, the DATAX model is used to map the RDF data model into Java objects.

A second possibility is that the `fipa:implementedBy` property includes code which is directly embedded as a (Java) script. The property `fipa:script` of the resource `fipa:Code` can be used these for purposes. Once again, conventions are needed to map the RDF data and the Java (script) model. For an example, see Section 3.3, .

3 Exchange of Rules Extensions

This module allows the expression and exchange of rules, based on the FIPA-RDF0 model.

3.1 Introduction

Using the `fipa-rdf1` language, agents can exchange knowledge about rules. An agent can inform another agent about one of its own "house" rules, but may also request to fire a particular rule on (a subset of) their knowledge base. In general, we leave it up to the implementer of the agent how to use the exchanged rules. The `fipa-rdf1` builds on top of the `fipa-rdf0` schemas, and provides extra schema information for expressing rules.

We will distinguish between two different approaches for dealing with rules:

1. Rules exchanged as XML/RDF encoded expressions.

2. Rules exchanged as pieces of programming code (scripts or Java classes).

3.2 Rules in XML/RDF

An RDF rule consists of two basic components: a *selection* part and a *manipulation* part, which applies to all RDF resources contained in the selection. To express the selection, an RDF notation for this purpose is chosen. To express the manipulation part, which allows to change property values of the selected resources, we will simply use the RDF data model itself.

In order to select parts of the RDF data resources, one can use an RDF query language. No real standards do exist at the moment, but various specifications are available which define how to query/select particular RDF resources including:

- RDF Query Specification (see [W3Crdfquery]), and,

- A Query and Inference Service for RDF (see [Decker98]).

The selection results will be put in an RDF container, identified by the property `fipa:selection-result` of the rule. The manipulation part will then give an RDF description for all resources contained in the container of the selection results. The following is an example of an RDF encoded rule:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#"
         xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf1#"
         xmlns:car="http://www.cars.org/schemas#"
         xmlns:rdfq="http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html">

  <fipa:Rule rdf:ID="categorizeCars1">
    <fipa:selection-result rdf:ID="speedycars"/>
      <fipa:selection>
        <rdfq:rdfquery>
          <rdfq:From eachResource="http://www.carshop.com/res/">
            <rdfq:Select>
              <rdfq:Condition>
                <rdfq:equals>
                  <rdfq:Property name="rdf:type"/>
                  <rdfq:String>
                    http://www.cars.org/schemas#Car
                  </rdfq:String>
                </rdfq:equals>
```

```
                <rdfq:greaterThan>
                  <rdfq:Property name="http://www.cars.org/schemas#speed"/>
                  <rdfq:Integer>200</rdfq:Integer>
                </rdfq:greaterThan>
              </rdfq:Condition>
            </rdfq:Select>
          </rdfq:From>
        </rdfq:rdfquery>
      </fipa:selection>
      <fipa:manipulation>
        <rdf:Description rdf:aboutEach="speedycars">
          <car:category>speed-car</car:category>
        </rdf:Description>
      </fipa:manipulation>
    </fipa:selection-result>
  </fipa:Rule>
</rdf:RDF>
```

In the above example, first all cars are selected from all resources contained in `http://www.carshop.com/res/` for which the maximum speed exceeds 200 (km/h). In the manipulation part, for all resources contained in the resulting collection, the value of the property `car:category` is set to `speed-car`.

3.3 Exchanging Rules as Programming Code

A rule is directly expressed as some piece of code (which presumably also selects nodes, and subsequently manipulates the RDF data). For this purpose, the property `fipa:implementedAs` is attached to the `fipa:Rule` class, as the property `implementedBy` was attached to a `fipa:Action` class.

The following example states that "for all cars for which the property speed exceeds 200 (km/h), the property category should be set to `race-car`:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#"
         xmlns:car="http://www.cars.org/schemas#"
         xmlns="http://www.fipa.org/schemas/fipa-rdf1#">

  <fipa:Rule rdf:ID="categorizeCars2">
    <fipa:implementedAs>
      <fipa:Code>
        <fipa:language>ECMAScript</fipa:language>
        <fipa:binding>3AP</fipa:binding>
        <fipa:script>
          NodeSelection selection = new NodeSelection("");
          Iterator it = selection.iterator();

          while (it.hasNext()) {
            Node n = (Node)it.next();

            if ((n.getProperty("speed").getValue() > 200) &
               (n.getProperty("type").getValue().
               equals("http://www.cars.org/schemas#Car")) {
                 n.getProperty("category").setValue("race-car");
            }
          }
        </fipa:script>
```

```
      </fipa:Code>
    </fipa:implementedAs>
  </fipa:Rule>
</rdf:RDF>
```

This script uses the 3AP APIs to map the RDF data with the Java object model.

3.4 Using Rules with FIPA Communicative Acts

An agent may request another agent to fire a specific rule to his knowledge base.

```
(request
  :sender i
  :receiver j
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#"
             xmlns="http://www.fipa.org/schemas/fipa-rdf1#">

      <FireRule>
        <rdf:type rdf:resource="http://www.fipa.org/schemas#Action"/>
        <argument rdf:resource="#categorizeCars2">
      </FireRule>
    </rdf:RDF> )
  :language fipa-rdf1 )
```

The rules engine will then have an impact on the properties of all car instances.

Another use is that an agent informs another agent about its (implicit) belief in the correctness of a rule:

```
(inform
  :sender i
  :receiver j
  :content (
    <?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/TR/REC-rdf-syntax#"
             xmlns="http://www.fipa.org/schemas/fipa-rdf1#">

      <fipa:Rule about="#categorizeCars2"/>
    </rdf:RDF> )
  :language fipa-rdf0)
```

The receiving agent may then decide to apply the rule (or not).

3.5 Further Remarks

In practice, the RDF content in a FIPA message may look quite verbose. However, this problem can be tackled in different ways:

- The RDF specification itself has been foreseen in a number of alternative 'abbreviated forms'.
- Binary encodings can be used instead, as defined by the XML Token specification (see [W3Cxml]).
- Some parts of the content can be defined in advance by unique XML identifiers (URIs) and then used in subsequent messages. This may be especially useful when the negotiation focuses only on one specific service parameter.

To support the latter mechanism of cross-referencing parts of the RDF content, we suggest the usage of the `query-ref` and `inform` (see [FIPA00046]) FIPA communicative acts.

4 Examples of Use

A number of companies and organisations in the FACTS project (see [FACTS]) have used FIPA RDF as content language for agent-based provisioning of virtual private networks.

4.1 RDF Schemas for FIPA RDF 0

The RDF schema needed for using `fipa-rdf0` (for expressing actions and propositions) is as follows:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">

  <rdfs:Class rdf:ID="Proposition">
    <rdfs:label xml:lang="en">proposition</rdfs:label>
    <rdfs:label xml:lang="fr">proposition</rdfs:label>
    <rdfs:subClassOf rdf:resource=
      "http://www.w3c.org/1999/02/22-rdf-syntax-ns#Statement"/>
    <rdfs:comment>This describes the set of propositions</rdfs:comment>
  </rdfs:Class>

  <rdfs:ConstraintProperty rdf:ID="belief">
    <rdfs:label xml:lang="en">belief</rdfs:label>
    <rdfs:label xml:lang="fr">acte</rdfs:label>
    <rdfs:domain rdf:resource="#Proposition"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>

  <rdfs:Class rdf:ID="Action">
    <rdfs:label xml:lang="en">action</rdfs:label>
    <rdfs:label xml:lang="fr">action</rdfs:label>
    <rdfs:subClassOf rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Resource"/>
    <rdfs:comment>This describes the set of actions</rdfs:comment>
  </rdfs:Class>

  <rdfs:ConstraintProperty rdf:ID="act">
    <rdfs:label xml:lang="en">act</rdfs:label>
    <rdfs:label xml:lang="fr">acte</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="actor">
    <rdfs:label xml:lang="en">actor</rdfs:label>
    <rdfs:label xml:lang="fr">acteur</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="argument">
    <rdfs:label xml:lang="en">argument</rdfs:label>
    <rdfs:label xml:lang="fr">argument</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>
```

```
  <rdfs:ConstraintProperty rdf:ID="done">
    <rdfs:label xml:lang="en">done</rdfs:label>
    <rdfs:label xml:lang="fr">fini</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="result">
    <rdfs:label xml:lang="en">result</rdfs:label>
    <rdfs:label xml:lang="fr">resultat</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="implementedBy">
    <rdfs:label xml:lang="en">implementedBy</rdfs:label>
    <rdfs:label xml:lang="fr">implemente par</rdfs:label>
    <rdfs:domain rdf:resource="#Action"/>
  </rdfs:ConstraintProperty>

  <rdfs:Class rdf:ID="Code">
    <rdfs:label xml:lang="en">code</rdfs:label>
    <rdfs:label xml:lang="fr">code</rdfs:label>
    <rdfs:comment>This describes the code implementation</rdfs:comment>
  </rdfs:Class>

  <rdfs:ConstraintProperty rdf:ID="language">
    <rdfs:label xml:lang="en">language</rdfs:label>
    <rdfs:label xml:lang="fr">langue</rdfs:label>
    <rdfs:domain rdf:resource="#Code"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="binding">
    <rdfs:label xml:lang="en">binding</rdfs:label>
    <rdfs:label xml:lang="fr">binding</rdfs:label>
    <rdfs:domain rdf:resource="#Code"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="code-uri">
    <rdfs:label xml:lang="en">code-uri</rdfs:label>
    <rdfs:label xml:lang="fr">code-uri</rdfs:label>
    <rdfs:domain rdf:resource="#Code"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="script">
    <rdfs:label xml:lang="en">script</rdfs:label>
    <rdfs:label xml:lang="fr">script</rdfs:label>
    <rdfs:domain rdf:resource="#Code"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Literal"/>
  </rdfs:ConstraintProperty>
</rdf:RDF>
```

4.2 RDF Schemas for FIPA RDF 1

The RDF schemas corresponding to `fipa-rdf1` are specified as follows (extending the above schemas):

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
         xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

  <rdfs:Class rdf:ID="Rule">
    <rdfs:label xml:lang="en">rule</rdfs:label>
    <rdfs:label xml:lang="fr">regle</rdfs:label>
  </rdfs:Class>

  <rdfs:ConstraintProperty rdf:ID="selection">
    <rdfs:comment>The selection part </rdfs:comment>
    <rdfs:domain rdf:resource="Rule"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="manipulation">
    <rdfs:comment>The manipulation part</rdfs:comment>
    <rdfs:domain rdf:resource="Rule"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="selection-result">
    <rdfs:comment>
      Identifies the container filled with selection results
    </rdfs:comment>
    <rdfs:domain rdf:resource="Rule"/>
    <rdfs:range rdf:resource=
      "http://www.w3c.org/TR/1999/PR-rdf-schema-19990303#Bag"/>
  </rdfs:ConstraintProperty>

  <rdfs:ConstraintProperty rdf:ID="implementedAs">
    <rdfs:label xml:lang="en">implemented as</rdfs:label>
    <rdfs:label xml:lang="fr">implemente comme</rdfs:label>
    <rdfs:domain rdf:resource="Rule"/>
  </rdfs:ConstraintProperty>
</rdf:RDF>
```

5 References

[DATAX] DATAX: Data Exchange in XML. 1999.
http://www.megginson.com/DATAX/

[Decker98] A Query and Inference Service for RDF, Decker, S, Brickley, D, Saarela, J and Angele, J. 1998.
http://www.ilrt.bris.ac.uk/discovery/rdf-dev/purls/papers/QL98-queryservice/

[FACTS] FIPA Agent Communication Technologies and Services (FACTS).
http://www.labs.bt.com/profsoc/facts/

[FIPA00046] FIPA Inform Communicative Act Specification. Foundation for Intelligent Physical Agents, 2000.
http://www.fipa.org/specs/fipa00046/

[FIPA00054] FIPA Query Ref Communicative Act Specification. Foundation for Intelligent Physical Agents, 2000.
http://www.fipa.org/specs/fipa00054/

[Melnik99] Generic Interoperability Framework (GINF) Working Paper, Melnik, S. Stanford University, 1999.

[W3Crdf] Status for Resource Description Framework (RDF) Model and Syntax Specification (Proposed Recommendation). World Wide Web Consortium, 1999.
http://www.w3.org/TR/REC-rdf-syntax/

[W3Crdfquery] RDF Query Specification (Technical Contribution). World Wide Web Consortium, 1998.
http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html

[W3Crdfsch] Resource Description Framework (RDF) Schema Specification 1.0 (Candidate Recommendation). World Wide Web Consortium, 2000.
http://www.w3.org/TR/rdf-schema/

[W3Cxml] Extensible Markup Language (XML) 1.0 Specification (Recommendation). World Wide Web Consortium, 1998.
http://www.w3c.org/TR/REC-xml/

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA SL Content Language Specification

Document title	FIPA SL Content Language Specification		
Document number	XC00008G	**Document source**	FIPA TC C
Document status	Experimental	**Date of this status**	2001/08/10
Supersedes	FIPA00003		
Contact	fab@fipa.org		
Change history			
2000/09/28	Approved for Experimental		
2001/08/10	Line numbering added		

Geneva, Switzerland

Foreword

The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. FIPA makes the results of its activities available to all interested parties and intends to contribute its results to the appropriate formal standards bodies.

The members of FIPA are individually and collectively committed to open competition in the development of agent-based applications, services and equipment. Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international organization without restriction. In particular, members are not bound to implement or use specific agent-based standards, recommendations and FIPA specifications by virtue of their participation in FIPA.

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Procedures for Technical Work. A complete overview of the FIPA specifications and their current status may be found in the FIPA List of Specifications. A list of terms and abbreviations used in the FIPA specifications may be found in the FIPA Glossary.

FIPA is a non-profit association registered in Geneva, Switzerland. As of January 2000, the 56 members of FIPA represented 17 countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found at http://www.fipa.org/.

Contents

1 Scope

This specification defines a concrete syntax for the FIPA Semantic Language (SL) content language. This syntax and its associated semantics are suggested as a candidate content language for use in conjunction with the FIPA Agent Communication Language (see [FIPA00037]). In particular, the syntax is defined to be a sub-grammar of the very general s-expression syntax specified for message content given in [FIPA00037].

This content language is included in the specification on an informative basis. It is not mandatory for any FIPA implementation to implement the computational mechanisms necessary to process all of the constructs in this language. However, FIPA SL is a general purpose representation formalism that may be suitable for use in a number of different agent domains.

2 Grammar FIPA SL Concrete Syntax

See *Section 6, Annex A — Syntax and Lexical* Notation for an explanation of the used syntactic notation.

```
Content                = "(" ContentExpression+ ")".

ContentExpression      = IdentifyingExpression
                       | ActionExpression
                       | Proposition.

Proposition            = Wff.

Wff                    = AtomicFormula
                       | "(" UnaryLogicalOp  Wff ")"
                       | "(" BinaryLogicalOp Wff Wff ")"
                       | "(" Quantifier      Variable Wff ")"
                       | "(" ModalOp         Agent Wff ")"
                       | "(" ActionOp        ActionExpression ")"
                       | "(" ActionOp        ActionExpression Wff ")".

UnaryLogicalOp         = "not".

BinaryLogicalOp        = "and"
                       | "or"
                       | "implies"
                       | "equiv".

AtomicFormula          = PropositionSymbol
                       | "(" BinaryTermOp    Term Term ")"
                       | "(" PredicateSymbol Term+ ")"
                       | "true"
                       | "false".

BinaryTermOp           = "="
                       | "\="
                       | ">"
                       | ">="
                       | "<"
                       | "=<"
                       | "member"
                       | "contains"
                       | "result".

Quantifier             = "forall"
                       | "exists".

ModalOp                = "B"
                       | "U"
                       | "PG"
                       | "I".

ActionOp               = "feasible"
                       | "done".

Term                   = Variable
                       | FunctionalTerm
                       | ActionExpression
                       | IdentifyingExpression
                       | Constant
                       | Sequence
                       | Set.
```

```
IdentifyingExpression  = "(" ReferentialOperator Term Wff ")".

ReferentialOperator    = "iota"
                       | "any"
                       | "all".

FunctionalTerm         = "(" "cons"         Term Term ")"
                       | "(" "first"        Term ")"
                       | "(" "rest"         Term ")"
                       | "(" "nth"          Term Term ")"
                       | "(" "append"       Term Term ")"
                       | "(" "union"        Term Term ")"
                       | "(" "intersection" Term Term ")"
                       | "(" "difference"   Term Term ")"
                       | "(" ArithmeticOp   Term Term ")"
                       | "(" FunctionSymbol Term* ")"
                       | "(" FunctionSymbol Parameter* ")".

Constant               = NumericalConstant
                       | String
                       | DateTime.

NumericalConstant      = Integer
                       | Float.

Variable               = VariableIdentifier.

ActionExpression       = "(" "action" Agent Term ")"
                       | "(" "|" ActionExpression ActionExpression ")"
                       | "(" ";" ActionExpression ActionExpression ")".

PropositionSymbol      = String.

PredicateSymbol        = String.

FunctionSymbol         = String.

Agent                  = Term.

Sequence               = "(" "sequence" Term* ")".

Set                    = "(" "set" Term* ")".

Parameter              = ParameterName ParameterValue.

ParameterValue         = Term.

ArithmeticOp           = "+"
                       | "-"
                       | "*"
                       | "/"
                       | "%".
```

2.1 Lexical Definitions

All white space, tabs, carriage returns and line feeds between tokens should be skipped by the lexical analyser. See *Section 6, Annex A — Syntax and Lexical Notation* for an explanation of the used notation.

```
String                 = Word
                       | StringLiteral.

Word                   = [~ "\0x00" - "\0x20", "(", ")", "#", "0" - "9", ":", "-", "?"]
                         [~ "\0x00" - "\0x20", "(", ")"]*.
```

```
196
197   ParameterName          = ":" String.
198
199   VariableIdentifier     = "?" String.
200
201   Sign                   = [ "+" , "-" ].
202
203   Integer                = Sign? DecimalLiteral+
204                          | Sign? "0" ["x", "X"] HexLiteral+.
205
206   Dot                    = "."
207
208   Float                  = Sign? FloatMantissa FloatExponent?
209                          | Sign? DecimalLiteral+ FloatExponent.
210
211   FloatMantissa          = DecimalLiteral+ Dot DecimalLiteral*
212                          | DecimalLiteral* Dot DecimalLiteral+.
213
214   FloatExponent          = Exponent Sign? DecimalLiteral+.
215
216   Exponent               = ["e","E"].
217
218   DecimalLiteral         = ["0" - "9"].
219
220   HexLiteral             = ["0" - "9", "A" - "F", "a" - "f"].
221
222   StringLiteral          = "\""( [~ "\""]
223                          | "\\\"" )*"\"".
224
225   DateTime               = Year Month Day "T" Hour Minute
226                            Second MilliSecond TypeDesignator?.
227
228   Year                   = DecimalLiteral DecimalLiteral DecimalLiteral DecimalLiteral.
229
230   Month                  = DecimalLiteral DecimalLiteral.
231
232   Day                    = DecimalLiteral DecimalLiteral.
233
234   Hour                   = DecimalLiteral DecimalLiteral.
235
236   Minute                 = DecimalLiteral DecimalLiteral.
237
238   Second                 = DecimalLiteral DecimalLiteral.
239
240   MilliSecond            = DecimalLiteral DecimalLiteral DecimalLiteral.
241
242   TypeDesignator         = ["a" - "z" , "A" – "Z"].
243
244
```

3 Notes on FIPA SL Semantics

This section contains explanatory notes on the intended semantics of the constructs introduced in above.

3.1 Grammar Entry Point: FIPA SL Content Expression

An FIPA SL content expression may be used as the content of an ACL message. There are three cases:

A proposition, which may be assigned a truth value in a given context. Precisely, it is a well-formed formula (Wff) using the rules described in the `Wff` production. A proposition is used in the `inform` communicative act (CA) and other CAs derived from it.

An action, which can be performed. An action may be a single action or a composite action built using the sequencing and alternative operators. An action is used as a content expression when the act is `request` and other CAs derived from it.

An identifying reference expression (IRE), which identifies an object in the domain. This is the Referential operator and is used in the `inform-ref` macro act and other CAs derived from it.

Other valid content expressions may result from the composition of the above basic cases. For instance, an action-condition pair (represented by an `ActionExpression` followed by a `Wff`) is used in the `propose` act; an action-condition-reason triplet (represented by an `ActionExpression` followed by two `Wffs`) is used in the `reject-proposal` act. These are used as arguments to some ACL CAs in [FIPA00037].

3.2 Well-Formed Formulas

A well-formed formula is constructed from an atomic formula, whose meaning will be determined by the semantics of the underlying domain representation or recursively by applying one of the construction operators or logical connectives described in the `Wff` grammar rule. These are:

`(not <Wff>)`
Negation. The truth value of this expression is false if `Wff` is true. Otherwise it is true.

`(and <Wff0> <Wff1>)`
Conjunction. This expression is true iff[1] well-formed formulae `Wff0` and `Wff1` are both true, otherwise it is false.

`(or <Wff0> <Wff1>)`
Disjunction. This expression is false iff well-formed formulae `Wff0` and `Wff1` are both false, otherwise it is true.

`(implies <Wff0> <Wff1>)`
Implication. This expression is true if either `Wff0` is false or alternatively if `Wff0` is true and `Wff1` is true. Otherwise it is false. The expression corresponds to the standard material implication connective `Wff0` `Wff1`.

`(equiv <Wff0> <Wff1>)`
Equivalence. This expression is true if either `Wff0` is true and `Wff1` is true, or alternatively if `Wff0` is false and `Wff1` is false. Otherwise it is false.

`(forall <variable> <Wff>)`
Universal quantification. The quantified expression is true if `Wff` is true for every value of value of the quantified variable.

`(exists <variable> <Wff>)`

[1] If and only if.

Existential quantification. The quantified expression is true if there is at least one value for the variable for which `Wff` is true.

`(B <agent> <expression>)`
Belief. It is true that `agent` believes that `expression` is true.

`(U <agent> <expression>)`
Uncertainty. It is true that `agent` is uncertain of the truth of `expression`. `Agent` neither believes `expression` nor its negation, but believes that `expression` is more likely to be true than its negation.

`(I <agent> <expression>)`
Intention. It is true that `agent` intends that `expression` becomes true and will plan to bring it about.

`(PG <agent> <expression>)`
Persistent goal. It is true that `agent` holds a persistent goal that `expression` becomes true, but will not necessarily plan to bring it about.

`(feasible <ActionExpression> <Wff>)`
It is true that `ActionExpression` (or, equivalently, some event) can take place and just afterwards `Wff` will be true.

`(feasible <ActionExpression>)`
Same as `(feasible <ActionExpression> true)`.

`(done <ActionExpression> <Wff>)`
It is true that `ActionExpression` (or, equivalently, some event) has just taken place and just before that `Wff` was true.

`(done <ActionExpression>)`
Same as `(done <ActionExpression> true)`.

3.3 Atomic Formula

The atomic formula represents an expression which has a truth value in the language of the domain of discourse. Three forms are defined:

a given propositional symbol may be defined in the domain language, which is either true or false,

two terms may or may not be equal under the semantics of the domain language, or,

some predicate is defined over a set of zero or more arguments, each of which is a term.

The FIPA SL representation does not define a meaning for the symbols in atomic formulae: this is the responsibility of the domain language representation and ontology. Several forms are defined:

`true false`
These symbols represent the true proposition and the false proposition.

`(= Term1 Term2)`
`Term1` and `Term2` denote the same object under the semantics of the domain.

`(\= Term1 Term2)`
`Term1` and `Term2` do not denote the same object under the semantics of the domain.

`(> Constant1 Constant2)`

The > operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes after the object denoted by `Constant2`, under the semantics of the domain.

`(>= Constant1 Constant2)`
The >= operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes after or is the same object as the object denoted by `Constant2`, under the semantics of the domain.

`(< Constant1 Constant2)`
The < operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes before the object denoted by `Constant2`, under the semantics of the domain.

`(=< Constant1 Constant2)`
The =< operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes before or is the same object as the object denoted by `Constant2`, under the semantics of the domain.

`(member Term Collection)`
The object denoted by `Term`, under the semantics of the domain, is a member of the collection (either a set or a sequence) denoted by `Collection` under the semantics of the domain.

`(contains Collection1 Collection2)`
If `Collection1` and `Collection2` denote sets, this proposition means the set denoted by `Collection1` contains the set denoted by `Collection2`. If the arguments are sequences, then the proposition means that all of the elements of the sequence denoted by `Collection2` appear in the same order in the sequence denoted by `Collection1`.

Other predicates may be defined over a set of arguments, each of which is a term, by using the `(PredicateSymbol Term+)` production.

The FIPA SL representation does not define a meaning for other symbols in atomic formulae: this is the responsibility of the domain language representation and the relative ontology.

3.4 Terms

Terms are either themselves atomic (constants and variables) or recursively constructed as a functional term in which a functor is applied to zero or more arguments. Again, FIPA SL only mandates a syntactic form for these terms. With small number of exceptions (see below), the meanings of the symbols used to define the terms are determined by the underlying domain representation.

Note that, as mentioned above, no legal well-formed expression contains a free variable, that is, a variable not declared in any scope within the expression. Scope introducing formulae are the quantifiers (`forall`, `exists`) and the reference operators `iota`, `any` and `all`. Variables may only denote terms, not well-formed formulae.

3.5 Referential Operators

3.5.1 Iota

`(iota <term> <formula>)`
The `iota` operator introduces a scope for the given expression (which denotes a term), in which the given identifier, which would otherwise be free, is defined. An expression containing a free variable is not a well-formed

FIPA SL expression. The expression `(iota x (P x))` may be read as "the `x` such that `P` [is true] of `x`". The `iota` operator is a constructor for terms which denote objects in the domain of discourse.

Formal Definition

A `iota` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such that T(KB) is the theory generated from KB by a given reasoning mechanism. Formally, (,) iff is a term that belongs to the set { : T(KB)} and is a singleton; or (,) is undefined if is not a singleton. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to . This implies that a failure occurs if no object or more than one object satisfies the condition specified in the `iota` operator.

Example 1

This example depicts an interaction between agent A and B that makes use of the `iota` operator, where agent A is supposed to have the following knowledge base KB={P(A), Q(1, A), Q(1, B)}.

```
(query-ref
  :sender (agent-idenfier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?x (p ?x)))
  :language FIPA-SL
  :reply-with query1)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B)
  :content
    ((= (iota ?x (p ?x)) a))
  :language FIPA-SL
  :in-reply-to query1)
```

The only object that satisfies proposition P(x) is a, therefore, the `query-ref` message is replied by the `inform` message as shown.

Example 2

This example shows another successful interaction but more complex than the previous one.

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?x (q ?x ?y)))
  :language FIPA-SL
  :reply-with query2)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((= (iota ?x (q ?x ?y)) 1))
  :language FIPA-SL
  :in-reply-to query2)
```

The most general substitutions such that Q(x, y) can be derived from KB are $_1$ {x/1, y/A} and $_2$ {x/1, y/B}. Therefore, the set { : T(KB)} {{x/1, y/A}x, {x/1, y/B}x } {1} is a singleton and hence `(iota ?x (q ?x ?y))` represents the object 1.

Example 3

Finally, this example shows an unsuccessful interaction using the `iota` operator. In this case, agent A cannot evaluate the `iota` expression and therefore a failure message is returned to agent B

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?y (q ?x ?y)))
  :language FIPA-SL
  :reply-with query3)

(failure
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((action (agent-identifier :name A)
             (inform-ref
              :sender (agent-identifier :name A)
              :receiver (set (agent-identifier :name B))
              :content
                 "((iota ?y (q ?x ?y)))"
              :language FIPA-SL
              :in-reply-to query3))
     more-than-one-answer)
  :language FIPA-SL
  :in-reply-to query3)
```

The most general substitutions that satisfy Q(x, y) are $_1$ {x/1, y/a} and $_2$ {x/1, y/b}, therefore, the set { : T(KB)} {{x/1, y/A}y, {x/1, y/B}y} {A, B}, which is not a singleton. This means that the `iota` expression used in this interaction is not defined.

3.5.2 Any

`(any <term> <formula>)`

The `any` operator is used to denote any object that satisfies the proposition represented by `formula`.

Formal Definition

An `any` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such that T(KB) is the theory generated from KB by a given reasoning mechanism. Formally, any(,) iff is a term that belongs to the set { : T(KB)}; or any(,) is undefined if is the empty set. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to .

This definition implies that failures only occur if there are no objects satisfying the condition specified as the second argument of the `any` operator.

Example 4

Assuming that agent A has the following knowledge base KB={P(A), Q(1, A), Q(1, B)}, this example shows a successful interaction with agent A using the `any` operator.

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((any (sequence ?x ?y) (q ?x ?y)))
  :language FIPA-SL
  :reply-with query1)

(inform
  :sender (agent-identifier :name A)
```

```
  :receiver (set (agent-identifier :name B))
  :content
    ((= (any (sequence ?x ?y) (q ?x ?y)) (sequence 1 a)))
  :language FIPA-SL
  :in-reply-to query1)
```

The most general substitutions such that Q(x, y) can be derived from KB are {x/1, y/A} and {x/1, y/B}, therefore { Sequence(x, y): Q(x, y) T(KB)}={Sequence(1, A), Sequence(1, B)}. Using this set, agent A chooses the first element of as the appropriate answer to agent B.

Example 5

This example shows an unsuccessful interaction with agent A, using the `any` operator.

```
(query-ref
  :sender   (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((any ?x (r ?x)))
  :language FIPA-SL
  :reply-with query2)

(failure
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((action (agent-identifier :name A)
             (inform-ref
               :sender (agent-identifier :name A)
               :receiver (set (agent-identifier :name B))
               :content
                 "((any ?x (r ?x)))"
               :language FIPA-SL
               :in-reply-to query2))
    (unknown-predicate r))
  :language FIPA-SL
  :in-reply-to query2)
```

Since agent A does not know the `r` predicate, the answer to the query that had been sent by agent B cannot be determined, therefore a failure message is sent to agent B from agent A. The failure message specifies the failure's reason (i.e., unknown-predicate `r`)

3.5.3 All

`(all <term> <formula>)`

The `all` operator is used to denote the set of all objects that satisfy the proposition represented by `formula`.

Formal Definition

An `all` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such that T(KB) is the theory generated from KB by a given reasoning mechanism. Formally, all(,) { : T(KB)}. Notice that all(,) may be a singleton or even an empty set. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to .

If no objects satisfy the condition specified as the second argument of the `all` operator, then the identifying expression denotes an empty set.

Example 6

Suppose agent A has the following knowledge base KB={P(A), Q(1, A), Q(1, B)}. This example shows a successful interaction between agent A and B that make use of the `all` operator.

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((all (sequence ?x ?y) (q ?x ?y)))
  :language FIPA-SL
  :reply-with query1)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    (( =   (all (sequence ?x ?y) (q ?x ?y)) (set(sequence 1 a)(sequence 1 b))))
  :language FIPA-SL
  :in-reply-to query1)
```

The set of the most general substitutions such that Q(x, y) can be derived from KB is {{x/1, y/A}, {x/1, y/B}}, therefore all(Sequence(x, y), Q(x, y)) {Sequence(1, A), Sequence(1, B)}.

Example 7

Following Example 6, if there is no possible answer to a query making use of the `all` operator, then the agent should return the empty set.

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((all ?x (q ?x c)))
  :language FIPA-SL
  :reply-with query2)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((= (all ?x (q ?x c))(set)))
  :language FIPA-SL
  :in-reply-to query2)
```

Since there is no possible substitution for x such that Q(x, C) can be derived from KB, then all(x, Q(x, c))={}. In this interaction the term `(set)` represents the empty set.

3.6 Functional Terms

A functional term refers to an object via a functional relation (referred by the `FunctionSymbol`) with other objects (that is, the terms or parameters), rather than using the direct name of that object, for example, `(fatherOf Jesus)` rather than `God`.

Two syntactical forms can be used to express a functional term. In the first form the functional symbol is followed by a list of terms that are the arguments of the function symbol. The semantics of the arguments is position-dependent, for example, `(divide 10 2)` where 10 is the dividend and 2 is the divisor. In the second form each argument is preceded by its name, for example, `(divide :dividend 10 :divisor 2)`. This second form is particularly appropriate to represent descriptions where the function symbol should be interpreted as the constructor of an object, while the parameters represent the attributes of the object.

The following is an example of an object, instance of a vehicle class:

```
(vehicle
```

```
  :colour red
  :max-speed 100
  :owner (Person
    :name Luis
    :nationality Portuguese))
```

Some ontologies may decide to give a description of some concepts only in one or both of these two forms, that is by specifying, or not, a default order to the arguments of each function in the domain of discourse. How this order is specified is outside the scope of this specification.

Functional terms can be constructed by a domain functor applied to zero or more terms. Besides domain functions, FIPA SL includes functional terms constructed from widely used functional operators and their arguments described in *Table 1*.

Operator	Example	Description

`+`
`-`
`/`
`%`
`*`
	`5 % 2`	Usual arithmetic operations.

`Union`	`(union ?s1 ?s2)`	Represents the union of two sets.
`Intersection`	`(intersection ?s1 ?s2)`	Represents the intersection of two sets.
`Difference`	`(difference ?s1 ?s2)`	Represents the set difference between `?s1` and `?s2`.
`First`	`(first ?seq)`	Represents the first element of a sequence.
`Rest`	`(rest ?seq)`	Represents sequence `?seq` except its first element.
`Nth`	`(nth 3 ?seq)`	Represents the nth element of a sequence.
`Cons`	`(cons a (sequence b c))`	If its second argument is a sequence, it represents the sequence that results of inserting its first argument in front of its second argument. If its second argument is a set, it represents the set that has all elements contained in its second argument plus its first argument.
`Append`	`(append ?seq (sequence c d))`	Represents the sequence that results of concatenating its first argument with its second argument.

Table 1: Functional Operators

3.7 Result Predicate

A common need is to determine the result of performing an action or evaluating a term. To facilitate this operation, a standard predicate `result`, of arity two, is introduced to the language. Result/2 has the declarative meaning that the result of evaluating a term, or equivalently of performing an action, encoded by the first argument term, is the second argument term. However, it is expected that this declarative semantics will be implemented in a more efficient, operational way in any given FIPA SL interpreter.

A typical use of the `result` predicate is with a variable scoped by `iota`, giving an expression whose meaning is, for example, "the x which is the result of agent `i` performing `act`":

```
(iota x (result (action i act) x)))
```

3.8 Actions and Action Expressions

Action expressions are a special subset of terms. An action itself is introduced by the keyword `action` and comprises the agent of the action (that is, an identifier representing the agent performing the action) and a term denoting the action which is [to be] performed.

Notice that a specific type of action is an ACL communicative act (CA). When expressed in FIPA SL, syntactically an ACL communicative act is an action where the term denotes the CA including all its parameters, as referred by the used ontology. Example 5 includes an example of an ACL CA, encoded as a `String`, whose content embeds another CA.

Two operators are used to build terms denoting composite CAs:

the sequencing operator (`;`) denotes a composite act in which the first action (represented by the first operand) is followed by the second action, and,

the alternative operator (`|`) denotes a composite act in which either the first action occurs, or the second, but not both.

3.9 Agent Identifiers

An agent is represented by referring to its name. The name is defined using the standard format from [FIPA00023].

3.10 Numerical Constants

The standard definitions for integers and floating point numbers are assumed. However, due to the necessarily unpredictable nature of cross-platform dependencies, agents should not make strong assumptions about the precision with which another agent is able to represent a given numerical value. FIPA SL assumes only 32-bit representations of both integers and floating point numbers. Agents should not exchange message contents containing numerical values requiring more than 32 bits to encode precisely, unless some prior arrangement is made to ensure that this is valid.

3.11 Date and Time Constants

Time tokens are based on [ISO8601], with extension for millisecond durations. If no type designator is given, the local time zone is then used. The type designator for UTC is the character `Z`; UTC is preferred to prevent time zone ambiguities. Note that years must be encoded in four digits. As an example, 8:30 am on 15th April, 1996 local time would be encoded as:

```
19960415T083000000
```

The same time in UTC would be:

```
19960415T083000000Z
```

4 Reduced Expressivity Subsets of FIPA SL

The FIPA SL definition given above is a very expressive language, but for some agent communication tasks it is unnecessarily powerful. This expressive power has an implementation cost to the agent and introduces problems of the decidability of modal logic. To allow simpler agents, or agents performing simple tasks, to do so with minimal computational burden, this section introduces semantic and syntactic subsets of the full FIPA SL content language for use by the agent when it is appropriate or desirable to do so. These subsets are defined by the use of profiles, that is, statements of restriction over the full expressive power of FIPA SL. These profiles are defined in increasing order of expressivity as FIPA-SL0, FIPA-SL1 and FIPA-SL2.

Note that these subsets of FIPA SL, with additional ontological commitments (that is, the definition of domain predicates and constants) are used in other FIPA specifications.

4.1 FIPA SL0: Minimal Subset

Profile 0 is denoted by the normative constant `FIPA-SL0` in the `:language` parameter of an ACL message. Profile 0 of FIPA SL is the minimal subset of the FIPA SL content language. It allows the representation of actions, the determination of the result a term representing a computation, the completion of an action and simple binary propositions. The following defines the FIPA SL0 grammar:

```
Content            = "(" ContentExpression+ ")".

ContentExpression  = ActionExpression
                   | Proposition.

Proposition        = Wff.

Wff                = AtomicFormula
                   | "(" ActionOp ActionExpression ")".

AtomicFormula      = PropositionSymbol
                   | "(" "result"        Term Term ")"
                   | "(" PredicateSymbol Term+ ")"
                   | "true"
                   | "false".

ActionOp           = "done".

Term               = Constant
                   | Set
                   | Sequence
                   | FunctionalTerm
                   | ActionExpression.

ActionExpression   = "(" "action" Agent Term ")".

FunctionalTerm     = "(" FunctionSymbol Term* ")"
                   | "(" FunctionSymbol Parameter* ")".

Parameter          =  ParameterName ParameterValue.

ParameterValue     = Term.

Agent              = Term.

FunctionSymbol     = String.

PropositionSymbol  = String.

PredicateSymbol    = String.
```

```
Constant           = NumericalConstant
                   | String
                   | DateTime.

Set                = "(" "set" Term* ")".

Sequence           = "(" "sequence" Term* ")".

NumericalConstant  = Integer
                   | Float.
```

The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL0.

4.2 FIPA SL1: Propositional Form

Profile 1 is denoted by the normative constant `FIPA-SL1` in the `:language` parameter of an ACL message. Profile 1 of FIPA SL extends the minimal representational form of FIPA SL0 by adding Boolean connectives to represent propositional expressions. The following defines the FIPA SL1 grammar:

```
Content              = "(" ContentExpression+ ")".

ContentExpression    = ActionExpression
                     | Proposition.

Proposition          = Wff.

Wff                  = AtomicFormula
                     | "(" UnaryLogicalOp  Wff ")"
                     | "(" BinaryLogicalOp Wff Wff ")"
                     | "(" ActionOp        ActionExpression ")".

UnaryLogicalOp       = "not".

BinaryLogicalOp      = "and"
                     | "or".

AtomicFormula        = PropositionSymbol
                     | "(" "result"        Term Term ")"
                     | "(" PredicateSymbol Term+ ")"
                     | "true"
                     | "false".

ActionOp             = "done".

Term                 = Constant
                     | Set
                     | Sequence
                     | FunctionalTerm
                     | ActionExpression.

ActionExpression     = "(" "action" Agent Term ")".

FunctionalTerm       = "(" FunctionSymbol Term* ")"
                     | "(" FunctionSymbol Parameter* ")".

Parameter            = ParameterName ParameterValue.

ParameterValue       = Term.

Agent                = Term.
```

```
FunctionSymbol        = String.

PropositionSymbol     = String.

PredicateSymbol       = String.

Constant              = NumericalConstant
                      | String
                      | DateTime.

Set                   = "(" "set" Term* ")".

Sequence              = "(" "sequence" Term* ")".

NumericalConstant     = Integer
                      | Float.
```

The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL1.

4.3 FIPA SL2: Decidability Restrictions

Profile 2 is denoted by the normative constant `FIPA-SL2` in the `:language` parameter of an ACL message. Profile 2 of FIPA SL allows first order predicate and modal logic, but is restricted to ensure that it must be decidable. Well-known effective algorithms exist that can derive whether or not an FIPA SL2 Wff is a logical consequence of a set of Wffs (for instance KSAT and Monadic). The following defines the FIPA SL2 grammar:

```
Content               = "(" ContentExpression+ ")".

ContentExpression     = IdentifyingExpression
                      | ActionExpression
                      | Proposition.

Proposition           = PrenexExpression.

Wff                   = AtomicFormula
                      | "(" UnaryLogicalOp  Wff ")"
                      | "(" BinaryLogicalOp Wff Wff ")"
                      | "(" ModalOp         Agent PrenexExpression ")"
                      | "(" ActionOp        ActionExpression ")"
                      | "(" ActionOp        ActionExpression PrenexExpression ")".

UnaryLogicalOp        = "not".

BinaryLogicalOp       = "and"
                      | "or"
                      | "implies"
                      | "equiv".

AtomicFormula         = PropositionSymbol
                      | "(" "="             Term Term ")"
                      | "(" "result"        Term Term ")"
                      | "(" PredicateSymbol Term+ ")"
                      | "true"
                      | "false".

PrenexExpression      = UnivQuantExpression
                      | ExistQuantExpression
                      | Wff.

UnivQuantExpression   = "(" "forall" Variable Wff ")"
                      | "(" "forall" Variable UnivQuantExpression ")"
                      | "(" "forall" Variable ExistQuantExpression ")".
```

```
ExistQuantExpression  = "(" "exists" Variable Wff ")"
                      | "(" "exists" Variable ExistQuantExpression ")".

Term                  =  Variable
                      | FunctionalTerm
                      | ActionExpression
                      | IdentifyingExpression
                      | Constant
                      | Sequence
                      | Set.

IdentifyingExpression = "(" ReferentialOp Term Wff ")".

ReferentialOp         = "iota"
                      | "any"
                      | "all".

FunctionalTerm        = "(" FunctionSymbol Term* ")"
                      | "(" FunctionSymbol Parameter* ")".

Parameter             = ParameterName ParameterValue.

ParameterValue        = Term.

ActionExpression      = "(" "action" Agent Term ")"
                      | "(" "|" ActionExpression ActionExpression ")"
                      | "(" ";" ActionExpression ActionExpression ")".

Variable              = VariableIdentifier.

Agent                 = Term.

FunctionSymbol        = String.

Constant              = NumericalConstant
                      | String
                      | DateTime.

ModalOp               = "B"
                      | "U"
                      | "PG"
                      | "I".

ActionOp              = "feasible"
                      | "done".

PropositionSymbol     = String.

PredicateSymbol       = String.

Set                   = "(" "set" Term* ")".

Sequence              = "(" "sequence" Term* ")".

NumericalConstant      = Integer
                       | Float.
```

The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL2.

The `Wff` production of FIPA SL2 no longer directly contains the logical quantifiers, but these are treated separately to ensure only prefixed quantified formulas, such as:

```
(forall ?x1
  (forall ?x2
    (exists ?y1
      (exists ?y2
        (Phi ?x1 ?x2 ?y1 ?y2)))))
```

Where `(Phi ?x1 ?x2 ?y1 ?y2)` does not contain any quantifier.

The grammar of FIPA SL2 still allows for quantifying-in inside modal operators. For example, the following formula is still admissible under the grammar:

```
(forall ?x1
  (or
    (B i (p ?x1))
    (B j (q ?x1))))
```

It is not clear that formulae of this kind are decidable. However, changing the grammar to express this context sensitivity would make the EBNF form above essentially unreadable. Thus, the following additional mandatory constraint is placed on well-formed content expressions using FIPA SL2:

Within the scope of an `SLModalOperator` only closed formulas are allowed, that is, formulas without free variables.

5 References

[FIPA00023] FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00023/

[FIPA00037] FIPA Agent Communication Language Overview. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00037/

[ISO8601] Date Elements and Interchange Formats, Information Interchange-Representation of Dates and Times. International Standards Organisation, 1998. http://www.iso.ch/cate/d15903.html

6 Annex A — Syntax and Lexical Notation

The syntax is expressed in standard EBNF format. For completeness, the notation is given in *Table 2.*

Grammar rule component	Example
Terminal tokens are enclosed in double quotes	`"("`
Non terminals are written as capitalised identifiers	`Expression`
Square brackets denote an optional construct	`[ "," OptionalArg ]`
Vertical bar denotes an alternative	`Integer | Real`
Asterisk denotes zero or more repetitions of the preceding expression	`Digit *`
Plus denotes one or more repetitions of the preceding expression	`Alpha +`
Parentheses are used to group expansions	`( A | B ) *`
Productions are written with the non-terminal name on the left-hand side, expansion on the right-hand side and terminated by a full stop	`AnonTerminal = "an expansion".`

Table 2: EBNF Rules

Some slightly different rules apply for the generation of lexical tokens. Lexical tokens use the same notation as above, with the exceptions noted in *Table 3*.

Lexical rule component	Example
Square brackets enclose a character set	`["a", "b", "c"]`
Dash in a character set denotes a range	`["a" - "z"]`
Tilde denotes the complement of a character set if it is the first character	`[~ "(", ")"]`
Post-fix question-mark operator denotes that the preceding lexical expression is optional (may appear zero or one times)	`["0" - "9"]? ["0" - "9"]`

Table 3: Lexical Rules

Ontology

In the discussion of speech acts, the sentence,

- I will meet you on Mars.

was presented as a speech act that in some sense (but not a logical sense) was false.

On the other hand, the sentence,

- I will meet you at the show.

makes sense.

How do we know whyone is nonsense and the other, sensible. Because we share an ontology which we have learned by "being in the world". We know what the words "Mars", and "show" refer to. Furthermore, we know certain relationships between the referents of these words. For example, we know quite well that we live on Earth which is a long way from Mars.

This shared ontolgy allows us to communicate with one another. The same idea is carried over to the worlds of artificial agents. For artificial agents to communicate they also must share ontologies.

What is an ontology?

Ontology 101 (pdf)

Protege-2000

Wine Ontology

Ontology Development 101: A Guide to Creating Your First Ontology

Natalya F. Noy and Deborah L. McGuinness
Stanford University, Stanford, CA, 94305
noy@smi.stanford.edu and dlm@ksl.stanford.edu

1 Why develop an ontology?

In recent years the development of ontologies—explicit formal specifications of the terms in the domain and relations among them (Gruber 1993)—has been moving from the realm of Artificial-Intelligence laboratories to the desktops of domain experts. Ontologies have become common on the World-Wide Web. The ontologies on the Web range from large taxonomies categorizing Web sites (such as on Yahoo!) to categorizations of products for sale and their features (such as on Amazon.com). The WWW Consortium (W3C) is developing the Resource Description Framework (Brickley and Guha 1999), a language for encoding knowledge on Web pages to make it understandable to electronic agents searching for information. The Defense Advanced Research Projects Agency (DARPA), in conjunction with the W3C, is developing DARPA Agent Markup Language (DAML) by extending RDF with more expressive constructs aimed at facilitating agent interaction on the Web (Hendler and McGuinness 2000). Many disciplines now develop standardized ontologies that domain experts can use to share and annotate information in their fields. Medicine, for example, has produced large, standardized, structured vocabularies such as SNOMED (Price and Spackman 2000) and the semantic network of the Unified Medical Language System (Humphreys and Lindberg 1993). Broad general-purpose ontologies are emerging as well. For example, the United Nations Development Program and Dun & Bradstreet combined their efforts to develop the UNSPSC ontology which provides terminology for products and services (www.unspsc.org).

An ontology defines a common vocabulary for researchers who need to share information in a domain. It includes machine-interpretable definitions of basic concepts in the domain and relations among them.

Why would someone want to develop an ontology? Some of the reasons are:

- To share common understanding of the structure of information among people or software agents
- To enable reuse of domain knowledge
- To make domain assumptions explicit
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge

Sharing common understanding of the structure of information among people or software agents is one of the more common goals in developing ontologies (Musen 1992; Gruber 1993). For example, suppose several different Web sites contain medical information or provide medical e-commerce services. If these Web sites share and publish the same underlying ontology of the terms they all use, then computer agents can extract and aggregate information from these different sites. The agents can use this aggregated information to answer user queries or as input data to other applications.

Enabling reuse of domain knowledge was one of the driving forces behind recent surge in ontology research. For example, models for many different domains need to represent the notion

of time. This representation includes the notions of time intervals, points in time, relative measures of time, and so on. If one group of researchers develops such an ontology in detail, others can simply reuse it for their domains. Additionally, if we need to build a large ontology, we can integrate several existing ontologies describing portions of the large domain. We can also reuse a general ontology, such as the UNSPSC ontology, and extend it to describe our domain of interest.

Making explicit domain assumptions underlying an implementation makes it possible to change these assumptions easily if our knowledge about the domain changes. Hard-coding assumptions about the world in programming-language code makes these assumptions not only hard to find and understand but also hard to change, in particular for someone without programming expertise. In addition, explicit specifications of domain knowledge are useful for new users who must learn what terms in the domain mean.

Separating the domain knowledge from the operational knowledge is another common use of ontologies. We can describe a task of configuring a product from its components according to a required specification and implement a program that does this configuration independent of the products and components themselves (McGuinness and Wright 1998). We can then develop an ontology of PC-components and characteristics and apply the algorithm to configure made-to-order PCs. We can also use the same algorithm to configure elevators if we "feed" an elevator component ontology to it (Rothenfluh et al. 1996).

Analyzing domain knowledge is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable when both attempting to reuse existing ontologies and extending them (McGuinness et al. 2000).

Often an ontology of the domain is not a goal in itself. Developing an ontology is akin to defining a set of data and their structure for other programs to use. Problem-solving methods, domain-independent applications, and software agents use ontologies and knowledge bases built from ontologies as data. For example, in this paper we develop an ontology of wine and food and appropriate combinations of wine with meals. This ontology can then be used as a basis for some applications in a suite of restaurant-managing tools: One application could create wine suggestions for the menu of the day or answer queries of waiters and customers. Another application could analyze an inventory list of a wine cellar and suggest which wine categories to expand and which particular wines to purchase for upcoming menus or cookbooks.

About this guide

We build on our experience using Protégé-2000 (Protege 2000), Ontolingua (Ontolingua 1997), and Chimaera (Chimaera 2000) as ontology-editing environments. In this guide, we use Protégé-2000 for our examples.

The wine and food example that we use throughout this guide is loosely based on an example knowledge base presented in a paper describing CLASSIC—a knowledge-representation system based on a description-logics approach (Brachman et al. 1991). The CLASSIC tutorial (McGuinness et al. 1994) has developed this example further. Protégé-2000 and other frame-based systems describe ontologies declaratively, stating explicitly what the class hierarchy is and to which classes individuals belong.

Some ontology-design ideas in this guide originated from the literature on object-oriented design (Rumbaugh et al. 1991; Booch et al. 1997). However, ontology development is different from designing classes and relations in object-oriented programming. Object-oriented programming centers primarily around methods on classes—a programmer makes design decisions based on the *operational* properties of a class, whereas an ontology designer makes these decisions based on the *structural* properties of a class. As a result, a class structure and relations among classes in

an ontology are different from the structure for a similar domain in an object-oriented program.

It is impossible to cover all the issues that an ontology developer may need to grapple with and we are not trying to address all of them in this guide. Instead, we try to provide a starting point; an initial guide that would help a new ontology designer to develop ontologies. At the end, we suggest places to look for explanations of more complicated structures and design mechanisms if the domain requires them.

Finally, there is no single correct ontology-design methodology and we did not attempt to define one. The ideas that we present here are the ones that we found useful in our own ontology-development experience. At the end of this guide we suggest a list of references for alternative methodologies.

2 What is in an ontology?

The Artificial-Intelligence literature contains many definitions of an ontology; many of these contradict one another. For the purposes of this guide an **ontology** is a formal explicit description of concepts in a domain of discourse (**classes** (sometimes called **concepts**)), properties of each concept describing various features and attributes of the concept (**slots** (sometimes called **roles** or **properties**)), and restrictions on slots (**facets** (sometimes called **role restrictions**)). An ontology together with a set of individual **instances** of classes constitutes a **knowledge base**. In reality, there is a fine line where the ontology ends and the knowledge base begins.

Classes are the focus of most ontologies. Classes describe concepts in the domain. For example, a class of wines represents all wines. Specific wines are instances of this class. The Bordeaux wine in the glass in front of you while you read this document is an instance of the class of Bordeaux wines. A class can have **subclasses** that represent concepts that are more specific than the superclass. For example, we can divide the class of all wines into red, white, and rosé wines. Alternatively, we can divide a class of all wines into sparkling and non-sparkling wines.

Slots describe properties of classes and instances: `Château Lafite Rothschild Pauillac` wine has a full body; it is produced by the `Château Lafite Rothschild` winery. We have two slots describing the wine in this example: the slot body with the value full and the slot maker with the value `Château Lafite Rothschild` winery. At the class level, we can say that instances of the class `Wine` will have slots describing their `flavor`, `body`, `sugar level`, the `maker` of the wine and so on.[1]

All instances of the class `Wine`, and its subclass Pauillac, have a slot `maker` the value of which is an instance of the class Winery (Figure 1). All instances of the class Winery have a slot `produces` that refers to all the wines (instances of the class Wine and its subclasses) that the winery produces.

In practical terms, developing an ontology includes:

- defining classes in the ontology,
- arranging the classes in a taxonomic (subclass–superclass) hierarchy,
- defining slots and describing allowed values for these slots,
- filling in the values for slots for instances.

We can then create a knowledge base by defining individual instances of these classes filling in specific slot value information and additional slot restrictions.

[1] We capitalize class names and start slot names with low-case letters. We also use `typewriter font` for all terms from the example ontology.

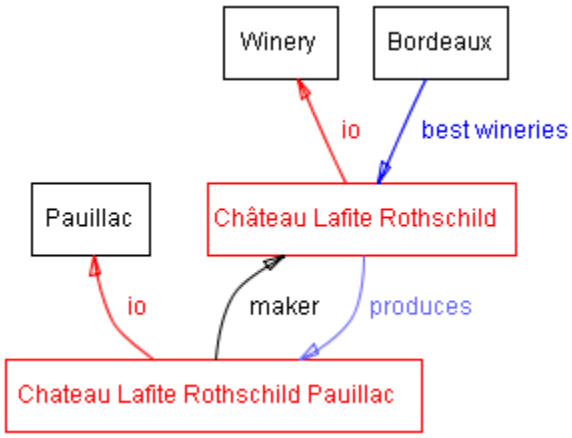


Figure 1. Some classes, instances, and relations among them in the wine domain. We used black for classes and red for instances. Direct links represent slots and internal links such as instance-of and subclass-of.

3 A Simple Knowledge-Engineering Methodology

As we said earlier, there is no one "correct" way or methodology for developing ontologies. Here we discuss general issues to consider and offer one possible process for developing an ontology. We describe an iterative approach to ontology development: we start with a rough first pass at the ontology. We then revise and refine the evolving ontology and fill in the details. Along the way, we discuss the modeling decisions that a designer needs to make, as well as the pros, cons, and implications of different solutions.

First, we would like to emphasize some fundamental rules in ontology design to which we will refer many times. These rules may seem rather dogmatic. They can help, however, to make design decisions in many cases.

1) *There is no one correct way to model a domain— there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.*
2) *Ontology development is necessarily an iterative process.*
3) *Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.*

That is, deciding what we are going to use the ontology for, and how detailed or general the ontology is going to be will guide many of the modeling decisions down the road. Among several viable alternatives, we will need to determine which one would work better for the projected task, be more intuitive, more extensible, and more maintainable. We also need to remember that an ontology is a model of reality of the world and the concepts in the ontology must reflect this reality. After we define an initial version of the ontology, we can evaluate and debug it by using it in applications or problem-solving methods or by discussing it with experts in the field, or both. As a result, we will almost certainly need to revise the initial ontology. This process of iterative design will likely continue through the entire lifecycle of the ontology.

Step 1. Determine the domain and scope of the ontology

We suggest starting the development of an ontology by defining its domain and scope. That is, answer several basic questions:

- What is the domain that the ontology will cover?
- For what we are going to use the ontology?
- For what types of questions the information in the ontology should provide answers?
- Who will use and maintain the ontology?

The answers to these questions may change during the ontology-design process, but at any given time they help limit the scope of the model.

Consider the ontology of wine and food that we introduced earlier. Representation of food and wines is the domain of the ontology. We plan to use this ontology for the applications that suggest good combinations of wines and food.

Naturally, the concepts describing different types of wines, main food types, the notion of a good combination of wine and food and a bad combination will figure into our ontology. At the same time, it is unlikely that the ontology will include concepts for managing inventory in a winery or employees in a restaurant even though these concepts are somewhat related to the notions of wine and food.

If the ontology we are designing will be used to assist in natural language processing of articles in wine magazines, it may be important to include synonyms and part-of-speech information for concepts in the ontology. If the ontology will be used to help restaurant customers decide which wine to order, we need to include retail-pricing information. If it is used for wine buyers in stocking a wine cellar, wholesale pricing and availability may be necessary. If the people who will maintain the ontology describe the domain in a language that is different from the language of the ontology users, we may need to provide the mapping between the languages.

Competency questions.

One of the ways to determine the scope of the ontology is to sketch a list of questions that a knowledge base based on the ontology should be able to answer, **competency questions** (Gruninger and Fox 1995). These questions will serve as the litmus test later: Does the ontology contain enough information to answer these types of questions? Do the answers require a particular level of detail or representation of a particular area? These competency questions are just a sketch and do not need to be exhaustive.

In the wine and food domain, the following are the possible competency questions:

- Which wine characteristics should I consider when choosing a wine?
- Is Bordeaux a red or white wine?
- Does Cabernet Sauvignon go well with seafood?
- What is the best choice of wine for grilled meat?
- Which characteristics of a wine affect its appropriateness for a dish?
- Does a bouquet or body of a specific wine change with vintage year?
- What were good vintages for Napa Zinfandel?

Judging from this list of questions, the ontology will include the information on various wine characteristics and wine types, vintage years—good and bad ones—classifications of foods that matter for choosing an appropriate wine, recommended combinations of wine and food.

Step 2. Consider reusing existing ontologies

It is almost always worth considering what someone else has done and checking if we can refine and extend existing sources for our particular domain and task. Reusing existing ontologies may be a requirement if our system needs to interact with other applications that have already committed to particular ontologies or controlled vocabularies. Many ontologies are already available in electronic form and can be imported into an ontology-development environment that you are using. The formalism in which an ontology is expressed often does not matter, since many knowledge-representation systems can import and export ontologies. Even if a knowledge-representation system cannot work directly with a particular formalism, the task of translating an ontology from one formalism to another is usually not a difficult one.

There are libraries of reusable ontologies on the Web and in the literature. For example, we can use the Ontolingua ontology library (http://www.ksl.stanford.edu/software/ontolingua/) or the DAML ontology library (http://www.daml.org/ontologies/). There are also a number of publicly available commercial ontologies (e.g., UNSPSC (www.unspsc.org), RosettaNet (www.rosettanet.org), DMOZ (www.dmoz.org)).

For example, a knowledge base of French wines may already exist. If we can import this knowledge base and the ontology on which it is based, we will have not only the classification of French wines but also the first pass at the classification of wine characteristics used to distinguish and describe the wines. Lists of wine properties may already be available from commercial Web sites such as www.wines.com that customers consider use to buy wines.

For this guide however we will assume that no relevant ontologies already exist and start developing the ontology from scratch.

Step 3. Enumerate important terms in the ontology

It is useful to write down a list of all terms we would like either to make statements about or to explain to a user. What are the terms we would like to talk about? What properties do those terms have? What would we like to say about those terms? For example, important wine-related terms will include `wine`, `grape`, `winery`, `location`, a wine's `color`, `body`, `flavor` and `sugar content`; different types of `food`, such as `fish` and `red meat`; subtypes of wine such as `white wine`, and so on. Initially, it is important to get a comprehensive list of terms without worrying about overlap between concepts they represent, relations among the terms, or any properties that the concepts may have, or whether the concepts are classes or slots.

The next two steps—developing the class hierarchy and defining properties of concepts (slots)—are closely intertwined. It is hard to do one of them first and then do the other. Typically, we create a few definitions of the concepts in the hierarchy and then continue by describing properties of these concepts and so on. These two steps are also the most important steps in the ontology-design process. We will describe them here briefly and then spend the next two sections discussing the more complicated issues that need to be considered, common pitfalls, decisions to make, and so on.

Step 4. Define the classes and the class hierarchy

There are several possible approaches in developing a class hierarchy (Uschold and Gruninger 1996):

- A **top-down** development process starts with the definition of the most general concepts in the domain and subsequent specialization of the concepts. For example, we can start with creating classes for the general concepts of `Wine` and `Food`. Then we specialize

the `Wine` class by creating some of its subclasses: `White wine`, `Red wine`, `Rosé wine`. We can further categorize the `Red wine` class, for example, into `Syrah`, `Red Burgundy`, `Cabernet Sauvignon`, and so on.

- A **bottom-up** development process starts with the definition of the most specific classes, the leaves of the hierarchy, with subsequent grouping of these classes into more general concepts. For example, we start by defining classes for `Pauillac` and `Margaux` wines. We then create a common superclass for these two classes—`Medoc`—which in turn is a subclass of `Bordeaux`.
- A **combination** development process is a combination of the top-down and bottom-up approaches: We define the more salient concepts first and then generalize and specialize them appropriately. We might start with a few top-level concepts such as `Wine`, and a few specific concepts, such as `Margaux` . We can then relate them to a middle-level concept, such as `Medoc`. Then we may want to generate all of the regional wine classes from France, thereby generating a number of middle-level concepts.

Figure 2 shows a possible breakdown among the different levels of generality.

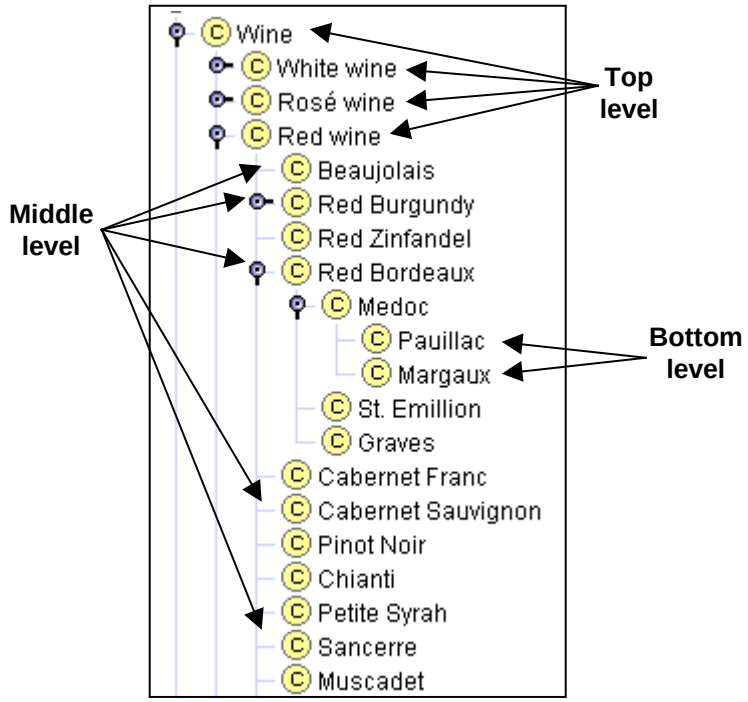


Figure 2. The different levels of the `Wine` taxonomy: `Wine` is the most general concept. `Red wine`, `White wine`, and `Rosé wine` are general top level concepts. `Pauillac` and `Margaux` are the most specific classes in the hierarchy (or the bottom level concepts).

None of these three methods is inherently better than any of the others. The approach to take depends strongly on the personal view of the domain. If a developer has a systematic top-down view of the domain, then it may be easier to use the top-down approach. The combination approach is often the easiest for many ontology developers, since the concepts "in the middle" tend to be the more descriptive concepts in the domain (Rosch 1978).

If you tend to think of wines by distinguishing the most general classification first, then the top-down approach may work better for you. If you'd rather start by getting grounded with specific examples, the bottom-up approach may be more appropriate.

Whichever approach we choose, we usually start by defining classes. From the list created in

Step 3, we select the terms that describe objects having independent existence rather than terms that describe these objects. These terms will be classes in the ontology and will become anchors in the class hierarchy.[2] We organize the classes into a hierarchical taxonomy by asking if by being an instance of one class, the object will necessarily (i.e., by definition) be an instance of some other class.

> *If a class A is a superclass of class B, then every instance of B is also an instance of A*

In other words, the class B represents a concept that is a "kind of" A.

For example, every Pinot Noir wine is necessarily a red wine. Therefore the `Pinot Noir` class is a subclass of the `Red Wine` class.

Figure 2 shows a part of the class hierarchy for the Wine ontology. Section 4 contains a detailed discussion of things to look for when defining a class hierarchy.

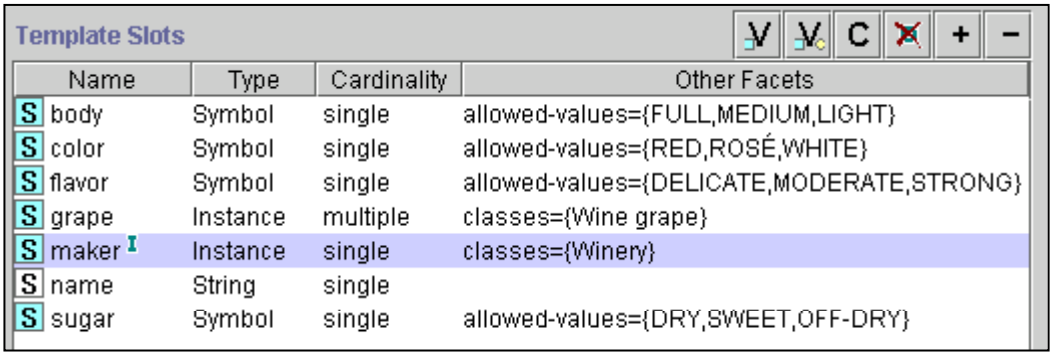


Figure 3. The slots for the class `Wine` and the facets for these slots. The "I" icon next to the `maker` slot indicates that the slot has an inverse (Section 5.1)

Step 5. Define the properties of classes—slots

The classes alone will not provide enough information to answer the competency questions from Step 1. Once we have defined some of the classes, we must describe the internal structure of concepts.

We have already selected classes from the list of terms we created in Step 3. Most of the remaining terms are likely to be properties of these classes. These terms include, for example, a wine's `color, body, flavor` and `sugar content` and `location` of a winery.

For each property in the list, we must determine which class it describes. These properties become slots attached to classes. Thus, the `Wine` class will have the following slots: `color`, body, `flavor`, and `sugar`. And the class `Winery` will have a `location` slot.

In general, there are several types of object properties that can become slots in an ontology:

- "intrinsic" properties such as the `flavor` of a wine;
- "extrinsic" properties such as a wine's `name,` and `area` it comes from;
- parts, if the object is structured; these can be both physical and abstract "parts" (e.g., the courses of a meal)

[2] We can also view classes as unary predicates—questions that have one argument. For example, "Is this object a wine?" Unary predicates (or classes) contrast with binary predicates (or slots)—questions that have two arguments. For example, "Is the flavor of this object strong?" "What is the flavor of this object?"

- relationships to other individuals; these are the relationships between individual members of the class and other items (e.g., the `maker` of a wine, representing a relationship between a wine and a winery, and the `grape` the wine is made from.)

Thus, in addition to the properties we have identified earlier, we need to add the following slots to the `Wine` class: `name`, `area`, `maker`, `grape`. Figure 3 shows the slots for the class `Wine`.

All subclasses of a class **inherit** the slot of that class. For example, all the slots of the class `Wine` will be inherited to all subclasses of Wine, including `Red Wine` and `White Wine`. We will add an additional slot, `tannin level` (low, moderate, or high), to the `Red Wine` class. The `tannin level` slot will be inherited by all the classes representing red wines (such as `Bordeaux` and `Beaujolais`).

A slot should be attached at the most general class that can have that property. For instance, `body` and `color` of a wine should be attached at the class `Wine`, since it is the most general class whose instances will have body and color.

Step 6. Define the facets of the slots

Slots can have different facets describing the value type, allowed values, the number of the values (cardinality), and other features of the values the slot can take. For example, the value of a `name` slot (as in "the name of a wine") is one string. That is, `name` is a slot with value type String. A slot `produces` (as in "a winery `produces` these wines") can have multiple values and the values are instances of the class Wine. That is, `produces` is a slot with value type `Instance` with `Wine` as allowed class.

We will now describe several common facets.

Slot cardinality

Slot cardinality defines how many values a slot can have. Some systems distinguish only between single cardinality (allowing at most one value) and multiple cardinality (allowing any number of values). A `body` of a wine will be a single cardinality slot (a wine can have only one body). Wines produced by a particular winery fill in a multiple-cardinality slot `produces` for a `Winery` class.

Some systems allow specification of a minimum and maximum cardinality to describe the number of slot values more precisely. Minimum cardinality of N means that a slot must have at least N values. For example, the `grape` slot of a Wine has a minimum cardinality of 1: each wine is made of at least one variety of grape. Maximum cardinality of M means that a slot can have at most M values. The maximum cardinality for the `grape` slot for single varietal wines is 1: these wines are made from only one variety of grape. Sometimes it may be useful to set the maximum cardinality to 0. This setting would indicate that the slot cannot have any values for a particular subclass.

Slot-value type

A value-type facet describes what types of values can fill in the slot. Here is a list of the more common value types:

- **String** is the simplest value type which is used for slots such as `name`: the value is a simple string
- **Number** (sometimes more specific value types of Float and Integer are used) describes slots with numeric values. For example, a `price` of a wine can have a value type Float

- **Boolean** slots are simple yes–no flags. For example, if we choose not to represent sparkling wines as a separate class, whether or not a wine is sparkling can be represented as a value of a Boolean slot: if the value is "true" ("yes") the wine is sparkling and if the value is "false" ("no") the wine is not a sparkling one.
- **Enumerated** slots specify a list of specific allowed values for the slot. For example, we can specify that the `flavor` slot can take on one of the three possible values: `strong`, `moderate`, and `delicate`. In Protégé-2000 the enumerated slots are of type `Symbol`.
- **Instance**-type slots allow definition of relationships between individuals. Slots with value type Instance must also define a list of allowed classes from which the instances can come. For example, a slot `produces` for the class `Winery` may have instances of the class `Wine` as its values.[3]

Figure 4 shows a definition of the slot `produces` at the class `Winery`.

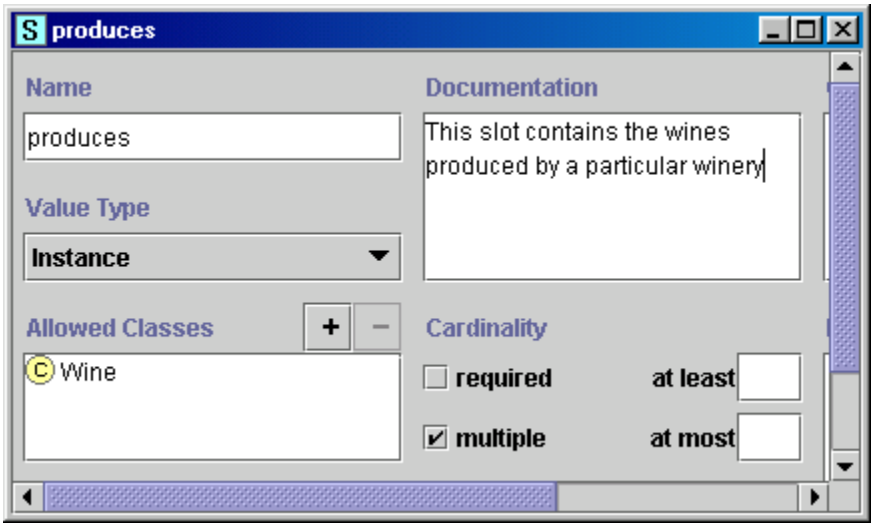


Figure 4. The definition of a slot `produces` that describes the wines produced by a winery. The slot has cardinality multiple, value type Instance, and the class `Wine` as the allowed class for its values.

Domain and range of a slot

Allowed classes for slots of type Instance are often called a **range** of a slot. In the example in Figure 4 the class `Wine` is the range of the `produces` slot. Some systems allow restricting the range of a slot when the slot is attached for a particular class.

The classes to which a slot is attached or a classes which property a slot describes, are called the **domain** of the slot. The `Winery` class is the domain of the `produces` slot. In the systems where we *attach* slots to classes, the classes to which the slot is attached usually constitute the domain of that slot. There is no need to specify the domain separately.

The basic rules for determining a domain and a range of a slot are similar:

> *When defining a domain or a range for a slot, find the most general classes or class that can be respectively the domain or the range for the slots .*
> *On the other hand, do not define a domain and range that is overly*

[3] Some systems just specify value type with a class instead of requiring a special statement of instance type slots.

general: all the classes in the domain of a slot should be described by the slot and instances of all the classes in the range of a slot should be potential fillers for the slot. Do not choose an overly general class for range (i.e., one would not want to make the range THING) but one would want to choose a class that will cover all fillers

Instead of listing all possible subclasses of the `Wine` class for the range of the `produces` slot, just list `Wine`. At the same time, we do not want to specify the range of the slot as THING—the most general class in an ontology.

In more specific terms:

If a list of classes defining a range or a domain of a slot includes a class and its subclass, remove the subclass.

If the range of the slot contains both the `Wine` class and the `Red Wine` class, we can remove the `Red Wine` from the range because it does not add any new information: The `Red Wine` is a subclass of `Wine` and therefore the slot range already implicitly includes it as well as all other subclasses of the `Wine` class.

If a list of classes defining a range or a domain of a slot contains all subclasses of a class A, but not the class A itself, the range should contain only the class A and not the subclasses.

Instead of defining the range of the slot to include `Red Wine`, `White Wine`, and `Rose Wine` (enumerating all the direct subclasses of `Wine`), we can limit the range to the class `Wine` itself.

If a list of classes defining a range or a domain of a slot contains all but a few subclasses of a class A, consider if the class A would make a more appropriate range definition.

In systems where attaching a slot to a class is the same as adding the class to the domain of the slot, the same rules apply to slot attachment: On the one hand, we should try to make it as general as possible. On the other hand, we must ensure that each class to which we attach the slot can indeed have the property that the slot represents. We can attach the `tannin level` slot to each of the classes representing red wines (e.g., `Bordeaux`, `Merlot`, `Beaujolais`, etc.). However, since all red wines have the tannin-level property, we should instead attach the slot to this more general class of `Red Wines`. Generalizing the domain of the `tannin level` slot further (by attaching it to the `Wine` class instead) would not be correct since we do not use tannin level to describe white wines for example.

Step 7. Create instances

The last step is creating individual instances of classes in the hierarchy. Defining an individual instance of a class requires (1) choosing a class, (2) creating an individual instance of that class, and (3) filling in the slot values. For example, we can create an individual instance `Chateau-Morgon-Beaujolais` to represent a specific type of Beaujolais wine. `Chateau-Morgon-Beaujolais` is an instance of the class `Beaujolais` representing all Beaujolais wines. This instance has the following slot values defined (Figure 5):

- Body: Light
- Color: Red
- Flavor: Delicate
- Tannin level: Low
- Grape: Gamay (instance of the `Wine grape` class)

- Maker: Chateau-Morgon (instance of the `Winery` class)
- Region: Beaujolais (instance of the `Wine-Region` class)
- Sugar: Dry

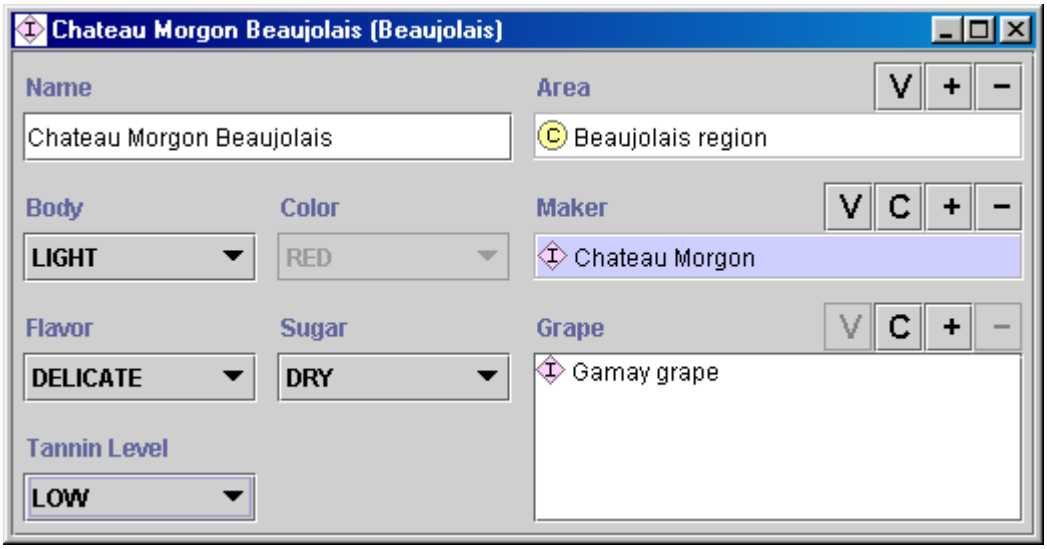


Figure 5. The definition of an instance of the `Beaujolais` class. The instance is `Chateaux Morgon Beaujolais` from the Beaujolais region, produced from the Gamay grape by the Chateau Morgon winery. It has a light body, delicate flavor, red color, and low tannin level. It is a dry wine.

4 Defining classes and a class hierarchy

This section discusses things to look out for and errors that are easy to make when defining classes and a class hierarchy (Step 4 from Section 3). As we have mentioned before, there is no single correct class hierarchy for any given domain. The hierarchy depends on the possible uses of the ontology, the level of the detail that is necessary for the application, personal preferences, and sometimes requirements for compatibility with other models. However, we discuss several guidelines to keep in mind when developing a class hierarchy. After defining a considerable number of new classes, it is helpful to stand back and check if the emerging hierarchy conforms to these guidelines.

4.1 Ensuring that the class hierarchy is correct

An "is-a" relation

The class hierarchy represents an "is-a" relation: a class A is a subclass of B if every instance of A is also an instance of B. For example, `Chardonnay` is a subclass of `White wine`. Another way to think of the taxonomic relation is as a "kind-of" relation: `Chardonnay` is a kind of `White wine`. A jetliner is a kind of an aircraft. Meat is a kind of food.

> *A subclass of a class represents a concept that is a "kind of" the concept that the superclass represents.*

A single wine is not a subclass of all wines

A common modeling mistake is to include both a singular and a plural version of the same concept in the hierarchy making the former a subclass of the latter. For example, it is wrong to define a class `Wines` and a class `Wine` as a subclass of `Wines`. Once you think of the hierarchy

as representing the "kind-of" relationship, the modeling error becomes clear: a single `Wine` is not a **kind of** `Wines`. The best way to avoid such an error is always to use either singular or plural in naming classes (see Section 6 for the discussion on naming concepts).

Transitivity of the hierarchical relations

A subclass relationship is transitive:

> *If B is a subclass of A and C is a subclass of B, then C is a subclass of A*

For example, we can define a class `Wine`, and then define a class `White wine` as a subclass of `Wine`. Then we define a class `Chardonnay` as a subclass of `White wine`. Transitivity of the subclass relationship means that the class `Chardonnay` is also a subclass of `Wine`. Sometimes we distinguish between direct subclasses and indirect subclasses. A **direct subclass** is the "closest" subclass of the class: there are no classes between a class and its direct subclass in a hierarchy. That is, there are no other classes in the hierarchy between a class and its direct superclass. In our example, `Chardonnay` is a direct subclass of `White wine` and is not a direct subclass of `Wine`.

Evolution of a class hierarchy

Maintaining a consistent class hierarchy may become challenging as domains evolve. For example, for many years, all Zinfandel wines were red. Therefore, we define a class of `Zinfandel` wines as a subclass of the `Red wine` class. Sometimes, however, wine makers began to press the grapes and to take away the color-producing aspects of the grapes immediately, thereby modifying the color of the resulting wine. Thus, we get "white zinfandel" whose color is rose. Now we need to break the `Zinfandel` class into two classes of zinfandel—`White zinfandel` and `Red zinfandel`—and classify them as subclasses of `Rose wine` and `Red wine` respectively.

Classes and their names

It is important to distinguish between a class and its name:

> *Classes represent concepts in the domain and not the words that denote these concepts.*

The name of a class may change if we choose a different terminology, but the term itself represents the objective reality in the world. For example, we may create a class `Shrimps`, and then rename it to `Prawns`—the class still represents the same concept. Appropriate wine combinations that referred to shrimp dishes should refer to prawn dishes. In more practical terms, the following rule should always be followed:

> *Synonyms for the same concept do not represent different classes*

Synonyms are just different names for a concept or a term. Therefore, we should **not** have a class called `Shrimp` and a class called `Prawn`, and, possibly a class called `Crevette`. Rather, there is one class, named either `Shrimp` or `Prawn`. Many systems allow associating a list of synonyms, translations, or presentation names with a class. If a system does not allow these associations, synonyms could always be listed in the class documentation.

Avoiding class cycles

We should avoid **cycles** in the class hierarchy. We say that there is a cycle in a hierarchy when some class A has a subclass B and at the same time B is a superclass of A. Creating such a cycle in a hierarchy amounts to declaring that the classes A and B are equivalent: all instances of A are instances of B and all instances of B are also instances of A. Indeed, since B is a subclass of A, all B's instances must be instances of the class A. Since A is a subclass of B, all A's instances

must also be instances of the class B.

4.2 Analyzing siblings in a class hierarchy

Siblings in a class hierarchy

Siblings in the hierarchy are classes that are direct subclasses of the same class (see Section 4.1).

> *All the siblings in the hierarchy (except for the ones at the root) must be at the same level of generality.*

For example, `White wine` and `Chardonnay` should not be subclasses of the same class (say, `Wine`). `White wine` is a more general concept than `Chardonnay`. Siblings should represent concepts that fall "along the same line" in the same way that same-level sections in a book are at the same level of generality. In that sense, requirements for a class hierarchy are similar to the requirements for a book outline.

The concepts at the root of the hierarchy however (which are often represented as direct subclasses of some very general class, such as `Thing`) represent major divisions of the domain and do not have to be similar concepts.

How many is too many and how few are too few?

There are no hard rules for the number of direct subclasses that a class should have. However, many well-structured ontologies have between two and a dozen direct subclasses. Therefore, we have the following two guidelines:

> *If a class has only one direct subclass there may be a modeling problem or the ontology is not complete.*
>
> *If there are more than a dozen subclasses for a given class then additional intermediate categories may be necessary.*

The first of the two rules is similar to a typesetting rule that bulleted lists should never have only one bullet point. For example, most of the red Burgundy wines are Côtes d'Or wines. Suppose we wanted to represent only this majority type of Burgundy wines. We could create a class `Red Burgundy` and then a single subclass `Cotes d'Or` (Figure 6a). However, if in our representation red Burgundy and Côtes d'Or wines are essentially equivalent (all red Burgundy wines are Côtes d'Or wines and all Côtes d'Or wines are red Burgundy wines), creating the `Cotes d'Or` class is not necessary and does not add any new information to the representation. If we were to include Côtes Chalonnaise wines, which are cheaper Burgundy wines from the region just South of Côtes d'Or, then we will create two subclasses of the `Burgundy` class: `Cotes d'Or` and `Cotes Chalonnaise` (Figure 6b).

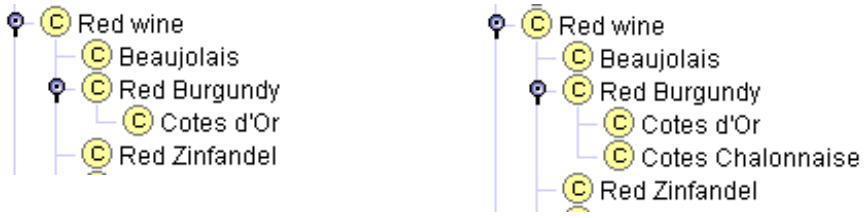


Figure 6. Subclasses of the `Red Burgundy` class. Having a single subclass of a class usually points to a problem in modeling.

Suppose now that we list all types of wines as direct subclasses of the `Wine` class. This list would then include such more general types of wine as Beaujolais and Bordeaux, as well as more specific types such as Paulliac and Margaux (Figure 6a). The class `Wine` has too many direct

subclasses and, indeed, for the ontology to reflect the different types of wine in a more organized manner, `Medoc` should be a subclass of `Bordeaux` and `Cotes d'Or` should be a subclass of `Burgundy`. Also having such intermediate categories as `Red wine` and `White wine` would also reflect the conceptual model of the domain of wines that many people have (Figure 6b).

However, if no natural classes exist to group concepts in the long list of siblings, there is no need to create artificial classes—just leave the classes the way they are. After all, the ontology is a reflection of the real world, and if no categorization exists in the real world, then the ontology should reflect that.

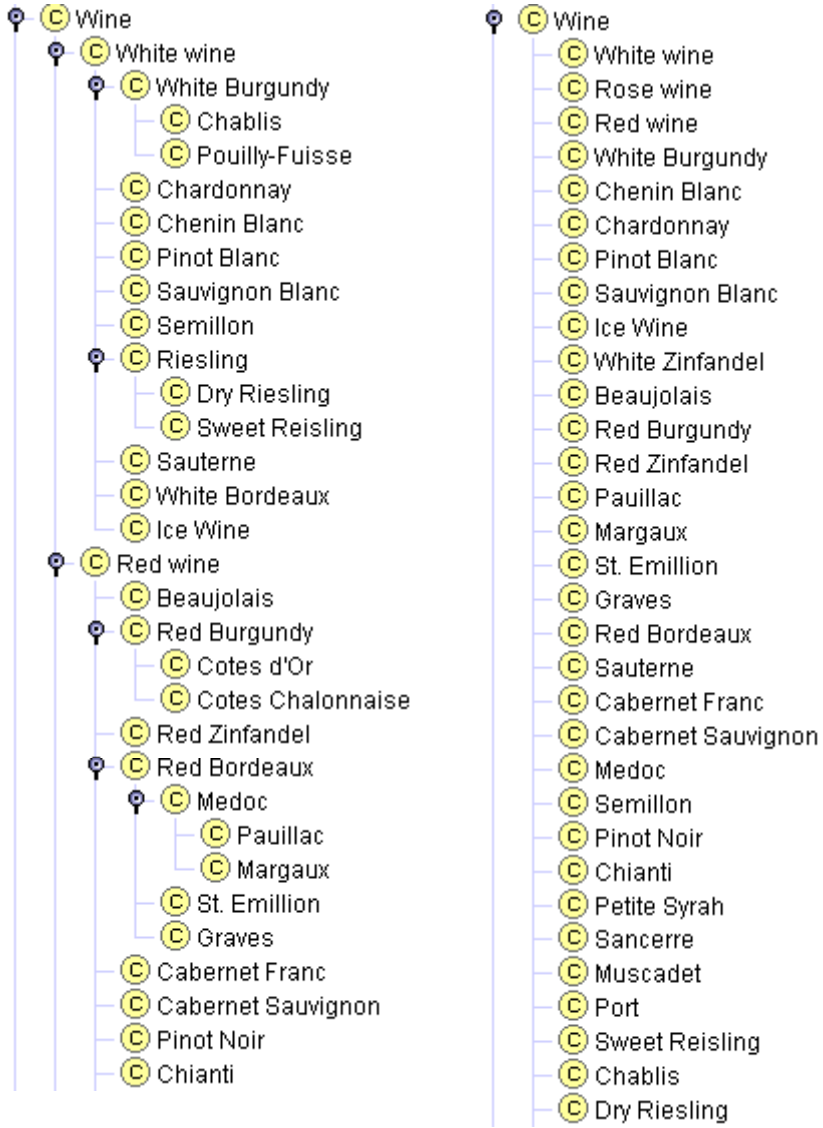


Figure 7. Categorizing wines. Having all the wines and types of wine versus having several levels of categorization.

4.3 Multiple inheritance

Most knowledge-representation systems allow **multiple inheritance** in the class hierarchy: a class can be a subclass of several classes. Suppose we would like to create a separate class of

dessert wines, the `Dessert wine` class. The Port wine is both a red wine and a dessert wine.[4] Therefore, we define a class `Port` to have two superclasses: `Red wine` and `Dessert wine`. All instances of the `Port` class will be instances of both the `Red wine` class and the `Dessert wine` class. The `Port` class will inherit its slots and their facets from both its parents. Thus, it will inherit the value SWEET for the slot `Sugar` from the `Dessert wine` class and the `tannin level` slot and the value for its color slot from the `Red wine` class.

4.4 When to introduce a new class (or not)

One of the hardest decisions to make during modeling is when to introduce a new class or when to represent a distinction through different property values. It is hard to navigate both an extremely nested hierarchy with many extraneous classes and a very flat hierarchy that has too few classes with too much information encoded in slots. Finding the appropriate balance though is not easy.

There are several rules of thumb that help decide when to introduce new classes in a hierarchy.

> *Subclasses of a class usually (1) have additional properties that the superclass does not have, or (2) restrictions different from those of the superclass, or (3) participate in different relationships than the superclasses*

Red wines can have different levels of tannin, whereas this property is not used to describe wines in general. The value for the sugar slot of the Dessert wine is SWEET, whereas it is not true of the superclass of the Dessert Wine class. Pinot Noir wines may go well with seafood whereas other red wines do not. In other words, we introduce a new class in the hierarchy usually only when there is something that we can say about this class that we cannot say about the superclass.

In practical terms, each subclass should either have new slots added to it, or have new slot values defined, or override some facets for the inherited slots.

However, sometimes it may be useful to create new classes even if they do not introduce any new properties.

> *Classes in terminological hierarchies do not have to introduce new properties*

For example, some ontologies include large reference hierarchies of common terms used in the domain. For example, an ontology underlying an electronic medical-record system may include a classification of various diseases. This classification may be just that—a hierarchy of terms, without properties (or with the same set of properties). In that case, it is still useful to organize the terms in a hierarchy rather than a flat list because it will (1) allow easier exploration and navigation and (2) enable a doctor to choose easily a level of generality of the term that is appropriate for the situation.

Another reason to introduce new classes without any new properties is to model concepts among which domain experts commonly make a distinction even though we may have decided not to model the distinction itself. Since we use ontologies to facilitate communication among domain experts and between domain experts and knowledge-based systems we would like to reflect the expert's view of the domain in the ontology.

Finally, we should not create subclasses of a class for each additional restriction. For example, we introduced the classes `Red wine`, `White wine`, and `Rose wine` because this distinction is a natural one in the wine world. We did not introduce classes for delicate wine,

[4] We chose to represent only red Ports in our ontology: white Ports do exist but they are extremely uncommon.

moderate wine, and so on. When defining a class hierarchy, our goal is to strike a balance between creating new classes useful for class organization and creating too many classes.

4.5 A new class or a property value?

When modeling a domain, we often need to decide whether to model a specific distinction (such as white, red, or rosé wine) as a property value or as a set of classes again depends on the scope of the domain and the task at hand.

Do we create a class `White wine` or do we simply create a class `Wine` and fill in different values for the slot `color`? The answer usually lies in the scope that we defined for the ontology. How important the concept of `White wine` is in our domain? If wines have only marginal importance in the domain and whether or not the wine is white does not have any particular implications for its relations to other objects, then we shouldn't introduce a separate class for white wines. For a domain model used in a factory producing wine labels, rules for wine labels of any color are the same and the distinction is not very important. Alternatively, for the representation of wine, food, and their appropriate combinations a red wine is very different from a white wine: it is paired with different foods, has different properties, and so on. Similarly, color of wine is important for the wines knowledge base that we may use to determine wine-tasting order. Thus, we create a separate class for `White wine`.

> *If the concepts with different slot values become restrictions for different slots in other classes, then we should create a new class for the distinction. Otherwise, we represent the distinction in a slot value.*

Similarly, our wine ontology has such classes as `Red Merlot` and `White Merlot`, rather than a single class for all Merlot wines: red Merlots and white Merlots are really different wines (made from the same grape) and if we are developing a detailed ontology of wine, this distinction is important.

> *If a distinction is important in the domain and we think of the objects with different values for the distinction as different kinds of objects, then we should create a new class for the distinction.*

Considering potential individual instances of a class may also be helpful in deciding whether or not to introduce a new class.

> *A class to which an individual instance belongs should not change often.*

Usually when we use extrinsic rather than intrinsic properties of concepts to differentiate among classes, instances of those classes will have to migrate often from one class to another. For example, `Chilled wine` should not be a class in an ontology describing wine bottles in a restaurant. The property c`hilled` should simply be an attribute of wine in a bottle since an instance of `Chilled wine` can easily cease being an instance of this class and then become an instance of this class again.

Usually numbers, colors, locations are slot values and do not cause the creation of new classes. Wine, however, is a notable exception since the color of the wine is so paramount to the description of wine.

For another example, consider the human-anatomy ontology. When we represent ribs, do we create a class for each of the "1st left rib", "2nd left rib", and so on? Or do we have a class Rib with slots for the order and the lateral position (left-right)?[5] If the information about each of the ribs that we represent in the ontology is significantly different, then we should indeed create a

[5] Here we assume that each anatomical organ is a class since we would also like to talk about "John's 1st left rib." Individual organs of existing people would be represented as individuals in our ontology.

class for each of the ribs. That is, if we want to represent details adjacency and location information (which is different for each rib) as well as specific functions that each rib playa and organs it protects, we want the classes. If we are modeling anatomy at a slightly lesser level of generality, and all ribs are very similar as far as our potential applications are concerned (we just talk about which rib is broken on the X-Ray without implications for other parts of the body), we may want to simplify our hierarchy and have just the class `Rib`, with two slots: `lateral position`, `order`.

4.6 An instance or a class?

Deciding whether a particular concept is a class in an ontology or an individual instance depends on what the potential applications of the ontology are. Deciding where classes end and individual instances begin starts with deciding what is the lowest level of granularity in the representation. The level of granularity is in turn determined by a potential application of the ontology. In other words, what are the most specific items that are going to be represented in the knowledge base? Going back to the competency questions we identified in Step 1 in Section 3, the most specific concepts that will constitute answers to those questions are very good candidates for individuals in the knowledge base.

> *Individual instances are the most specific concepts represented in a knowledge base.*

For example, if we are only going to talk about pairing wine with food we will not be interested in the specific physical bottles of wine. Therefore, such terms as `Sterling Vineyards Merlot` are probably going to be the most specific terms we use. In other words, the Wine class is a collection not of individual bottles of wines but rather of the specific wines produced by specific wineries. Therefore, `Sterling Vineyards Merlot` would be an instance in the knowledge base.

On the other hand, if we would like to maintain an inventory of wines in the restaurant in addition to the knowledge base of good wine-food pairings, individual bottles of each wine may become individual instances in our knowledge base.

Similarly, if we would like to record different properties for each specific vintage of the `Sterling Vineyards Merlot,` then the specific vintage of the wine is an instance in a knowledge base and `Sterling Vineyards Merlot` is a class containing instances for all its vintages.

Another rule can "move" some individual instances into the set of classes:

> *If concepts form a natural hierarchy, then we should represent them as classes*

Consider the wine regions. Initially, we may define main wine regions, such as France, United States, Germany, and so on, as classes and specific wine regions within these large regions as instances. For example, `Bourgogne region` is an instance of the `French region` class. However, we would also like to say that the `Cotes d'Or region` is a `Bourgogne region`. Therefore, `Bourgogne region` must be a class (in order to have subclasses or instances). However, making `Bourgogne region` a class and `Cotes d'Or region` an instance of `Bourgogne region` seems arbitrary: it is very hard to clearly distinguish which regions are classes and which are instances. Therefore, we define all wine regions as classes. Protégé-2000 allows users to specify some classes as Abstract, signifying that the class cannot have any direct instances. In our case, all region classes are abstract (Figure 8).

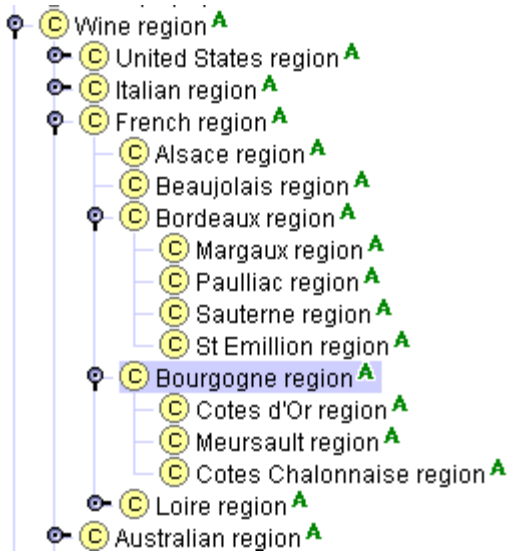


Figure 8. Hierarchy of wine regions. The "A" icons next to class names indicate that the classes are abstract and cannot have any direct instances.

The same class hierarchy would be incorrect if we omitted the word "region" from the class names. We cannot say that the class `Alsace` is a subclass of the class `France`: Alsace is not a kind of France. However, Alsace region is a kind of a French region.

Only classes can be arranged in a hierarchy—knowledge-representation systems do not have a notion of sub-instance. Therefore, if there is a natural hierarchy among terms, such as in terminological hierarchies from Section 4.2, we should define these terms as classes even though they may not have any instances of their own.

4.7 Limiting the scope

As a final note on defining a class hierarchy, the following set of rules is always helpful in deciding when an ontology definition is complete:

> *The ontology should not contain all the possible information about the domain: you do not need to specialize (or generalize) more than you need for your application (at most one extra level each way).*

For our wine and food example, we do not need to know what paper is used for the labels or how to cook shrimp dishes.

Similarly,

> *The ontology should not contain all the possible properties of and distinctions among classes in the hierarchy.*

In our ontology, we certainly do not include all the properties that a wine or food could have. We represented the most salient properties of the classes of items in our ontology. Even though wine books would tell us the size of grapes, we have not included this knowledge. Similarly, we have not added all relationships that one could imagine among all the terms in our system. For example, we do not include relationships such as `favorite wine` and `favorite food` in the ontology just to allow a more complete representation of all of the interconnections between the terms we have defined.

The last rule also applies to establishing relations among concepts that we have already included in the ontology. Consider an ontology describing biology experiments. The ontology will likely contain a concept of `Biological organisms`. It will also contain a concept of an

`Experimenter` performing an experiment (with his name, affiliation, etc.). It is true that an experimenter, as a person, also happens to be a biological organism. However, we probably should not incorporate this distinction in the ontology: for the purposes of this representation an experimenter is not a biological organism and we will probably never conduct experiments on the experimenters themselves. If we were representing everything we can say about the classes in the ontology, an `Experimenter` would become a subclass of `Biological Organism`. However, we do not need to include this knowledge for the foreseeable applications. In fact, including this type of additional classification for existing classes actually hurts: now an instance of an Experimenter will have slots for weight, age, species, and other data pertaining to a biological organism, but absolutely irrelevant in the context of describing an experiment. However, we should record such design decision in the documentation for the benefit of the users who will be looking at this ontology and who may not be aware of the application we had in mind. Otherwise, people intending to reuse the ontology for other applications may try to use experimenter as a subclass of person without knowing that the original modeling did not include that fact.

4.8 Disjoint subclasses

Many systems allow us to specify explicitly that several classes are **disjoint**. Classes are disjoint if they cannot have any instances in common. For example, the `Dessert wine` and the `White wine` classes in our ontology are *not* disjoint: there are many wines that are instances of both. The `Rothermel Trochenbierenauslese Riesling` instance of the `Sweet Riesling` class is one such example. At the same time, the `Red wine` and the `White wine` classes are disjoint: no wine can be simultaneously red and white. Specifying that classes are disjoint enables the system to validate the ontology better. If we declare the `Red wine` and the `White wine` classes to be disjoint and later create a class that is a subclass of both `Riesling` (a subclass of `White wine`) and `Port` (a subclass of `Red wine`), a system can indicate that there is a modeling error.

5 Defining properties—more details

In this section we discuss several more details to keep in mind when defining slots in the ontology (Step 5 and Step 6 in Section 3). Mainly, we discuss inverse slots and default values for a slot.

5.1 Inverse slots

A value of a slot may depend on a value of another slot. For example, if a `wine` was `produced by` a `winery`, then the `winery produces` that `wine`. These two relations, `maker` and `produces`, are called **inverse relations**. Storing the information "in both directions" is redundant. When we know that a wine is produced by a winery, an application using the knowledge base can always infer the value for the inverse relation that the winery produces the wine. However, from the knowledge-acquisition perspective it is convenient to have both pieces of information explicitly available. This approach allows users to fill in the wine in one case and the winery in another. The knowledge-acquisition system could then automatically fill in the value for the inverse relation insuring consistency of the knowledge base.

Our example has a pair of inverse slots: the `maker` slot of the `Wine` class and the `produces` slot of the `Winery` class. When a user creates an instance of the `Wine` class and fills in the value for the `maker` slot, the system automatically adds the newly created instance to the

`produces` slot of the corresponding `Winery` instance. For instance, when we say that Sterling Merlot is produced by the Sterling Vineyard winery, the system would automatically add Sterling Merlot to the list of wines that the Sterling Vineyard winery produces. (Figure 9).

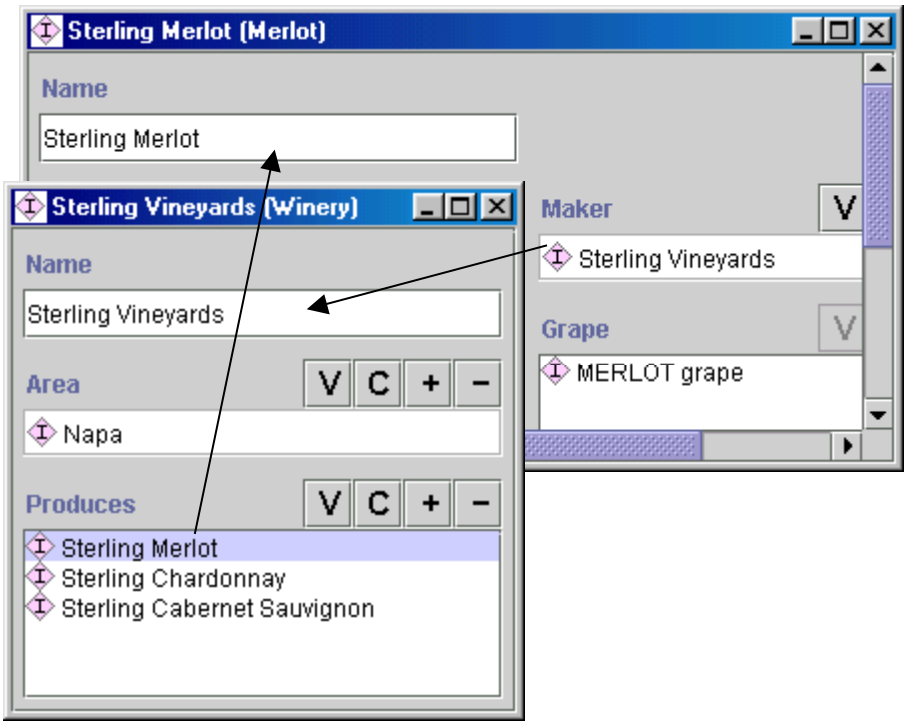


Figure 9. Instances with inverse slots. The slot `produces` for the class `Winery` is an inverse of the slot `maker` for the class `Wine`. Filling in one of the slots triggers an automatic update of the other.

5.2 Default values

Many frame-based systems allow specification of default values for slots. If a particular slot value is the same for most instances of a class, we can define this value to be a **default value** for the slot. Then, when each new instance of a class containing this slot is created, the system fills in the default value automatically. We can then change the value to any other value that the facets will allow. That is, default values are there for convenience: they do not enforce any new restrictions on the model or change the model in any way.

For example, if the majority of wines we are going to discuss are full-bodied wines, we can have "full" as a default value for the body of the wine. Then, unless we say otherwise, all wines we define would be full-bodied.

Note that this is different from **slot values**. Slot values cannot be changed. For example, we can say that the slot `sugar` has value SWEET for the `Dessert wine` class. Then all the subclasses and instances of the `Dessert wine` class will have the SWEET value for the slot `sugar`. This value cannot be changed in any of the subclasses or instances of the class.

6 What's in a name?

Defining naming conventions for concepts in an ontology and then strictly adhering to these conventions not only makes the ontology easier to understand but also helps avoid some common modeling mistakes. There are many alternatives in naming concepts. Often there is no particular reason to choose one or another alternative. However, we need to

Define a naming convention for classes and slots and adhere to it.

The following features of a knowledge representation system affect the choice of naming conventions:

- Does the system have the same name space for classes, slots, and instances? That is, does the system allow having a class and a slot with the same name (such as a class `winery` and a slot `winery`)?
- Is the system case-sensitive? That is, does the system treat the names that differ only in case as different names (such as `Winery` and `winery`)?
- What delimiters does the system allow in the names? That is, can names contain spaces, commas, asterisks, and so on?

Protégé-2000, for example, maintains a single name space for all its frames. It is case-sensitive. Thus, we cannot have a class `winery` and a slot `winery`. We can, however, have a class `Winery` (note the upper-case) and a slot `winery`. CLASSIC, on the other hand, is not case sensitive and maintains different name spaces for classes, slots, and individuals. Thus, from a system perspective, there is no problem in naming both a class and a slot `Winery`.

6.1 Capitalization and delimiters

First, we can greatly improve the readability of an ontology if we use consistent capitalization for concept names. For example, it is common to capitalize class names and use lower case for slot names (assuming the system is case-sensitive).

When a concept name contains more than one word (such as `Meal course`) we need to delimit the words. Here are some possible choices.

- Use Space: `Meal course` (many systems, including Protégé, allow spaces in concept names).
- Run the words together and capitalize each new word: `MealCourse`
- Use an underscore or dash or other delimiter in the name: `Meal_Course`, `Meal_course`, `Meal-Course`, `Meal-course`. (If you use delimiters, you will also need to decide whether or not each new word is capitalized)

If the knowledge-representation system allows spaces in names, using them may be the most intuitive solution for many ontology developers. It is however, important to consider other systems with which your system may interact. If those systems do not use spaces or if your presentation medium does not handle spaces well, it can be useful to use another method.

6.2 Singular or plural

A class name represents a collection of objects. For example, a class `Wine` actually represents all wines. Therefore, it could be more natural for some designers to call the class `Wines` rather than `Wine`. No alternative is better or worse than the other (although singular for class names is used more often in practice). However, whatever the choice, it should be consistent throughout the whole ontology. Some systems even require their users to declare in advance whether or not they are going to use singular or plural for concept names and do not allow them to stray from that choice.

Using the same form all the time also prevents a designer from making such modeling mistakes as creating a class `Wines` and then creating a class `Wine` as its subclass (see Section 4.1).

6.3 Prefix and suffix conventions

Some knowledge-base methodologies suggest using prefix and suffix conventions in the names to distinguish between classes and slots. Two common practices are to add a `has-` or a suffix `–of` to slot names. Thus, our slots become `has-maker` and `has-winery` if we chose the `has-` convention. The slots become `maker-of` and `winery-of` if we chose the `of-` convention. This approach allows anyone looking at a term to determine immediately if the term is a class or a slot. However, the term names become slightly longer.

6.4 Other naming considerations

Here are a few more things to consider when defining naming conventions:

- Do not add strings such as "class", "property", "slot", and so on to concept names.

It is always clear form the context whether the concept is a class or a slot, for example. In addition is you use different naming conventions for classes and slots (say, capitalization and no capitalization respectively), the name itself would be indicative of what the concept is.

- It is usually a good idea to avoid abbreviations in concept names (that is, use `Cabernet Sauvignon` rather than `Cab`)
- Names of direct subclasses of a class should either all include or not include the name of the superclass. For example, if we are creating two subclasses of the `Wine` class to represent red and white wines, the two subclass names should be either `Red Wine` and `White Wine` or `Red` and `White`, but not `Red Wine` and `White`.

7 Other Resources

We have used Protégé-2000 as an ontology-developing environment for our examples. Duineveld and colleagues (Duineveld et al. 2000) describe and compare a number of other ontology-development environments.

We have tried to address the very basics of ontology development and have not discussed many of the advanced topics or alternative methodologies for ontology development. Gómez-Pérez (Gómez-Pérez 1998) and Uschold (Uschold and Gruninger 1996) present alternative ontology-development methodologies. The Ontolingua tutorial (Farquhar 1997) discusses some formal aspects of knowledge modeling.

Currently, researchers emphasize not only ontology development, but also ontology analysis. As more ontologies are generated and reused, more tools will be available to analyze ontologies. For example, Chimaera (McGuinness et al. 2000) provides diagnostic tools for analyzing ontologies. The analysis that Chimaera performs includes both a check for logical correctness of an ontology and diagnostics of common ontology-design errors. An ontology designer may want to run Chimaera diagnostics over the evolving ontology to determine the conformance to common ontology-modeling practices.

8 Conclusions

In this guide, we have described an ontology-development methodology for declarative frame-based systems. We listed the steps in the ontology-development process and addressed the complex issues of defining class hierarchies and properties of classes and instances. However, after following all the rules and suggestions, one of the most important things to remember is the following: *there is no single correct ontology for any domain*. Ontology design is a creative process and no two ontologies designed by different people would be the same. The potential

applications of the ontology and the designer's understanding and view of the domain will undoubtedly affect ontology design choices. "The proof is in the pudding"—we can assess the quality of our ontology only by using it in applications for which we designed it.

Acknowledgments

Protégé-2000 (http://protege.stanford.edu) was developed by Mark Musen's group at Stanford Medical Informatics. We generated some of the figures with the OntoViz plugin to Protégé-2000. We imported the initial version of the wine ontology from the Ontolingua ontology library (http://www.ksl.stanford.edu/software/ontolingua/) which in turn used a version published by Brachman and colleagues (Brachman et al. 1991) and distributed with the CLASSIC knowledge representation system. We then modified the ontology to present conceptual-modeling principles for declarative frame-based ontologies. Ray Fergerson's and Mor Peleg's extensive comments on earlier drafts greatly improved this paper.

References

Booch, G., Rumbaugh, J. and Jacobson, I. (1997). *The Unified Modeling Language user guide*: Addison-Wesley.

Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A. and Borgida, A. (1991). Living with CLASSIC: When and how to use KL-ONE-like language. *Principles of Semantic Networks*. J. F. Sowa, editor, Morgan Kaufmann**:** 401-456.

Brickley, D. and Guha, R.V. (1999). Resource Description Framework (RDF) Schema Specification. Proposed Recommendation, World Wide Web Consortium**:** http://www.w3.org/TR/PR-rdf-schema.

Chimaera (2000). Chimaera Ontology Environment. www.ksl.stanford.edu/software/chimaera

Duineveld, A.J., Stoter, R., Weiden, M.R., Kenepa, B. and Benjamins, V.R. (2000). WonderTools? A comparative study of ontological engineering tools. *International Journal of Human-Computer Studies* **52**(6): 1111-1133.

Farquhar, A. (1997). Ontolingua tutorial. http://ksl-web.stanford.edu/people/axf/tutorial.pdf

Gómez-Pérez, A. (1998). Knowledge sharing and reuse. *Handbook of Applied Expert Systems*. Liebowitz, editor, CRC Press.

Gruber, T.R. (1993). A Translation Approach to Portable Ontology Specification. *Knowledge Acquisition* **5**: 199-220.

Gruninger, M. and Fox, M.S. (1995). Methodology for the Design and Evaluation of Ontologies. In: *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI-95*, Montreal.

Hendler, J. and McGuinness, D.L. (2000). The DARPA Agent Markup Language. *IEEE Intelligent Systems* **16**(6): 67-73.

Humphreys, B.L. and Lindberg, D.A.B. (1993). The UMLS project: making the conceptual connection between users and the information they need. *Bulletin of the Medical Library Association* **81**(2): 170.

McGuinness, D.L., Abrahams, M.K., Resnick, L.A., Patel-Schneider, P.F., Thomason, R.H., Cavalli-Sforza, V. and Conati, C. (1994). Classic Knowledge Representation System Tutorial. http://www.bell-labs.com/project/classic/papers/ClassTut/ClassTut.html

McGuinness, D.L., Fikes, R., Rice, J. and Wilder, S. (2000). An Environment for Merging and Testing Large Ontologies. *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*. A. G. Cohn, F. Giunchiglia and B. Selman, editors. San Francisco, CA, Morgan Kaufmann Publishers.

McGuinness, D.L. and Wright, J. (1998). Conceptual Modeling for Configuration: A Description Logic-based Approach. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing - special issue on Configuration*.

Musen, M.A. (1992). Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research* **25**: 435-467.

Ontolingua (1997). Ontolingua System Reference Manual. http://www-ksl-svc.stanford.edu:5915/doc/frame-editor/index.html

Price, C. and Spackman, K. (2000). SNOMED clinical terms. *BJHC&IM-British Journal of Healthcare Computing & Information Management* **17**(3): 27-31.

Protege (2000). The Protege Project. http://protege.stanford.edu

Rosch, E. (1978). Principles of Categorization. *Cognition and Categorization*. R. E. and B. B. Lloyd, editors. Hillside, NJ, Lawrence Erlbaum Publishers**:** 27-48.

Rothenfluh, T.R., Gennari, J.H., Eriksson, H., Puerta, A.R., Tu, S.W. and Musen, M.A. (1996). Reusable ontologies, knowledge-acquisition tools, and performance systems: PROTÉGÉ-II solutions to Sisyphus-2. *International Journal of Human-Computer Studies* **44**: 303-332.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991). *Object-oriented modeling and design*. Englewood Cliffs, New Jersey: Prentice Hall.

Uschold, M. and Gruninger, M. (1996). Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review* **11**(2).

JADE Administrative Tutorials

JADE comes in a zip file. Just unzip it, preserving the directory structure. On Windows systems you usually put it in c:\jade.

The Administrative Guide. (version 2.5)

This is the main reference for running JADE. It is quite detailed and somewhat terse. To supplement this guide, here aresome step by step tutorials.

Running JADE with one (MAIN) container.

Using more than one container

Running multiple JADE platforms.

Using the HTTP Message Transport Protocol.

JADE ADMINISTRATOR'S GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

Last update: 29-January-2002. JADE 2.5

Authors: Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT)
Giovanni Rimassa (University of Parma)

JADE - Java Agent DEvelopment Framework is a framework to develop multi-agent systems in compliance with the FIPA specifications. JADE successfully passed the 1st FIPA interoperability test in Seoul (Jan. 99) and the 2nd FIPA interoperability test in London (Apr. 01).

TABLE OF CONTENTS

1 INTRODUCTION

This administrator's guide describes how to install and launch JADE. It is complemented by the HTML documentation available in the directory jade/doc and the JADE Programmer's Guide. If and where conflict arises between what is reported in the HTML documentation and this guide, preference should be given to the HTML documentation that is updated more frequently.

2 RUNNING THE AGENT PLATFORM

2.1 Software requirements

The only software requirement to execute the system is the Java Run Time Environment version 1.2.

In order to build the system the JDK1.2 is sufficient because pre-built IDL stubs and Java parser classes are included with the JADE source distribution. Those users, who wish to regenerate IDL stubs and Java parser classes, should also have the JavaCC parser generator (version 0.8pre or version 1.1; available from http://www.metamata.com), and the IDL to Java compiler (available from the Sun Developer Connection). Notice that the old idltojava compiler available with JDK1.2 generates wrong code and it should never be used, instead the new idlj compiler, that is distributed with JDK1.3, should be used.

2.2 Getting the software

All the software is distributed under the LGPL license limitations and it can be downloaded from the JADE web site http://jade.cselt.it/. Five compressed files are available:

1. The source code of JADE
2. The source code of the examples
3. The documentation, including the javadoc of the JADE API and this programmer's guide
4. The binary of JADE, i.e. the jar files with all the Java classes
5. A full distribution with all the previous files

2.3 Running JADE from the binary distribution

Having uncompressed the archive file, a directory tree is generated whose root is `jade` and with a `lib` subdirectory. This subdirectory contains some JAR files that have to be added to the `CLASSPATH` environment variable.

Having set the classpath, the following command can be used to launch the main container of the platform. The main container is composed of the DF agent, the AMS agent, and the RMI registry (that is used by JADE for intra-platform communication).

```
java jade.Boot [options] [AgentSpecifier list]
```

Additional agent containers can be then launched on the same host, or on remote hosts, that connect themselves with the main container of the Agent Platform, resulting in a distributed system that seems a single Agent Platform from the outside.

An Agent Container can be started using the command:

```
java jade.Boot –container [options] [AgentSpecifier list]
```

An alternative way of launching JADE is to use the following command, that does not need to set the CLASSPATH:

```
java –jar lib\jade.jar –nomtp [options] [AgentSpecifier list]
```

Remind to use the "–nomtp" option, otherwise an exception will be thrown because the library iiop.jar is not found.

2.3.1 Command line syntax

The full EBNF syntax of the command line is the following, where common rules apply for the token definitions:

```
java jade.Boot Option* AgentSpecifier*

Option         = "-container"
               | "-host" HostName
               | "-port" PortNumber
               | "-name" PlatformName
               | "-gui"
               | "-mtp" ClassName "(" Argument* ")"
                   (";" ClassName "(" Argument* ")")*
               | "-nomtp"
               | "-aclcodec" ClassName (";" ClassName )*
               | "-nomobility"
               | "-version"
               | "-help"
               | "-conf" FileName?

ClassName      = PackageName? Word

PackageName    = (Word ".")+

Argument       = Word | Number | String

HostName       = Word ( "." Word )*

PortNumber     = Number

AgentSpecifier = AgentName ":" ClassName ("(" Argument* ")")?

AgentName      = Word

PlatformName   = Word
```

2.3.2 Options available from the command line

-container specifies that this instance of JADE is a container and, as such, that it must join with a main-container (by default this option is unselected)

-host specifies the host name where the RMI registry should be created (for the main-

container) / located (for the ordinary containers); its value is defaulted to localhost. This option can also be used when launching the main-container in order to override the value of localhost; **a typical example of this kind of usage is to include the full domain of the host (e.g. *–host kim.cselt.it* when the localhost would have returned just *'kim'*) such that the main-container can be contacted even from outside the local domain.**

-port this option allows to specify the port number where the RMI registry should be created (for the main-container) / located (for the ordinary containers). By default the port number 1099 is used.

-name this option specifies the symbolic name to be used as the platform name; this option will be considered only in the case of a main container; the default is to generate a unique name from the values of the main container's host name and port number. Please note that this option is strongly discouraged since uniqueness of the HAP is not enforced. This might result in non-unique agent names.

-gui specifies that the RMA (Remote Monitoring Agent) GUI of JADE should be launched (by default this option is unselected)

-mtp specifies a list of external Message Transport Protocols to be activated on this container (by default the JDK1.2 IIOP is activated on the main-container and no MTP is activated on the other containers)

-nomtp has precedence over -mtp and overrides it. It should be used to override the default behaviour of the main-container (by default the -nomtp option unselected)

-aclcodec By default all messages are encoded by the String-based ACLCodec. This option allows to specify a list of additional ACLCodec that will become available to the agents of the launched container in order to encode/decode messages. JADE will provide automatically to use these codec when agents set the right value in the field *aclRepresentation* of the Envelope of the sent/received ACLMessages. Look at the FIPA specifications for the standard names of these codecs

-nomobility disable the mobility and cloning support in the launched container. In this way the container will not accept requests for agent migration or agent cloning, option that might be useful to enhance the level of security for the host where this container is running. Notice that the platform can include both containers where mobility is enabled and containers where it is disabled. In this case an agent that tries to move from/to the containers where mobility is disabled will die because of a Runtime Exception.
Notice that, even if this option was selected, the container would still be able to launch new agents (e.g. via the RMA GUI) if their class can be reached via the local CLASSPATH.
By default this option is unselected.

-version print on standard output the versioning information of JADE (by default this option is unselected)

-help print on standard output this help information (by default this option is unselected)

-conf if no filename is specified after this option, then a graphical interface is displayed that allows to load/save all the JADE configuration parameters from a file. If a filename is specified, instead, then all the options specified in that file are used to launch JADE. By default this option is not selected

2.3.3 Launching agents from the command line

A list of agents can be launched directly from the command line. As described above, the `[AgentSpecifier list]` part of the command is a sequence of strings separated by a space.

Each string is broken into three parts. The first substring (delimited by the colon ':' character) is taken as the agent name; the remaining substring after the colon (ended with a space or with an open parenthesis) is the name of the Java class implementing the agent. The Agent Container will dynamically load this class. Finally, a list of string arguments can be passed delimited between parentheses.

For example, a string `Peter:myAgent` means "create a new agent named Peter whose implementation is an object of class `myAgent`". The name of the class must be fully qualified, (e.g. `Peter:myPackage.myAgent`) and will be searched for according to `CLASSPATH` definition.

Another example is the string `Peter:myAgent("today is raining" 123)` that means "create a new agent named Peter whose implementation is an object of class `myAgent` and pass an array of two arguments to its constructor: the first is the string `today is raining` and the second is the string 123". Notice that, according to the Java convention, the quote symbols have been removed and the number is still a string.

2.3.4 Example

First of all set the `CLASSPATH` to include the JAR files in the `lib` subdirectory and the current directory. For instance, for Windows 9x/NT use the following command:

```
set CLASSPATH=%CLASSPATH%;.;c:\jade\lib\jade.jar;
c:\jade\lib\jadeTools.jar;c:\jade\lib\Base64.jar;c:\jade\lib
\iiop.jar
```

Execute the following command to start the main-container of the platform. Let's suppose that the hostname of this machine is "kim.cselt.it"

```
prompt> java jade.Boot –gui
```

Execute the following command to start an agent container on another machine, by telling it to join the Agent Platform running on the host "kim.cselt.it", and start one agent (you must download and compile the examples agents to do that):

```
prompt> java jade.Boot -host kim.cselt.it -container
          sender1:examples.receivers.AgentSender
```

where "sender1" is the name of the agent, while `examples.receivers.AgentSender` is the code that implements the agent.

Execute the following command on a third machine to start another agent container telling it to join the Agent Platform, called "facts" running on the host "kim.cselt.it", and then start two agents.

```
prompt> java jade.Boot –host kim.cselt.it –container
          receiver2:examples.receivers.AgentReceiver
          sender2:examples.receivers.AgentSender
```

where the agent named sender2 is implemented by the class `examples.receivers.AgentSender`, while the agent named receiver2 is implemented by the class `examples.receivers.AgentReceiver`.

2.4 Building JADE from the source distribution

If you downloaded JADE in source form and want to compile it, you basically have two methods: either you use the provided makefiles (for GNU make), or you run the Win32 .BAT files that you find in the root directory of the package. Of course, using makefiles yields more flexibility because they just build what is needed; JADE makefiles have been tested under Sun Solaris 7 with JDK 1.2.0 and under Linux under JDK 1.2.2 RC4 and JDK 1.3. The batch files have been tested under Windows NT 4.0 and under Windows 95, both with JDK 1.2.2 or JDK1.3

2.4.1 Building the JADE framework

If you use the makefiles, just type:

```
make all
```

in the root directory; if you use the batch files, type

```
makejade
```

in the root directory. Beware that the batch file will not be able to check whether IDL stubs and parser classes already exist, so either you have idltojava and JavaCC installed, or you comment out them in the batch file.

You will end up with all JADE classes in a `classes` subdirectory. You can add that directory to your `CLASSPATH` and make sure that everything is OK by running JADE, as described in the previous section.

2.4.2 Building JADE libraries

With makefiles, type

```
make lib
```

With batch files, type

```
makelib
```

This will remove the content of the `classes` directory and will create some JAR files in the `lib` directory. These JAR files are just the same you get from the binary distribution. See section 2.3 for a description on how to run JADE when you have built the JAR files. Beware that, with both makefiles and batches, you must first build the classes and then the libraries, or you will end up with empty JAR files.

2.4.3 Building JADE HTML documentation

With makefiles, type

`make doc`

With batch files, type

`makedoc`

You will end up with Javadoc generated HTML pages, integrated within the overall documentation. Beware that the Programmer's Guide is a PDF file that cannot be generated at your site, but you must download it (it is, of course, in the JADE documentation distribution).

2.4.4 Building JADE examples and demo application

If you downloaded the examples/demo archive and have unpacked it within the same source tree, you will have to set your `CLASSPATH` to contain either the `classes` directory or the JAR files in the `lib` directory, depending on your JADE distribution, and then type:

`make examples`

with makefiles, or

`makeexamples`

with batch files.

In order to compile the Jess-based example, it is necessary to have the JESS system and to set the CLASSPATH to include it. The example can be compiled by typing:

`make jessexample`

with makefiles, or

`makejessexample`

with batch files.

2.4.5 Cleaning up the source tree

If you type

`make clean`

with makefiles, or if you type

`clean`

with batch files, you will remove all generated files (classes, HTML pages, JAR files, etc.) from the source tree. If you use makefiles, you will find some other make targets you can use. Feel free to try them, especially if you are modifying JADE source code, but be aware that these other make targets are for internal use only, so they have not been documented.

2.5 Support for inter-platform messaging with plug-in Message Transport Protocols

The FIPA 2000 specification proposes a number of different *Message Transport Protocols* (*MTP*s for short) over which ACL messages can be delivered in a compliant way.

JADE comprises a framework to write and deploy multiple *MTP*s in a flexible way. An implementation of a FIPA compliant MTP can be compiled separately and put in a JAR file of its own; the code will be dynamically loaded when an endpoint of that MTP is activated. Moreover, every JADE container can have any number of active MTPs, so that the platform administrator can choose whatever topology he or she wishes.

JADE performs message routing for both incoming and outgoing messages, using a single-hop routing table that requires direct visibility among containers.

When a new MTP is activated on a container, the JADE platform gains a new address that is added to the list in the platform profile (that can be obtained from the AMS using the action `get-description`). Moreover, the new address is added to all the `ams-agent-description` objects contained within the AMS knowledge base.

2.5.1 Command line options for MTP management

When a JADE container is started, it is possible to activate one ore more communication endpoints on it, using suitable command line options. The `–mtp` option activates a new communication endpoint on a container, and must be given the name of the class that provides the MTP functionality. If the MTP supports activation on specific addresses, then the address URL can be given right after the class name, enclosed in brackets. If multiple MTPs are to be activated, they can be listed together using commas as separators.

For example, the following option activates an IIOP endpoint on a default address.

```
-mtp jade.mtp.iiop.MessageTransportProtocol
```

The following option activates an IIOP endpoint that uses an ORBacus-based[1] IIOP MTP on a fixed, given address.

```
-mtp
orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cselt.it:1234/jade)
```

The following option activates two endpoints that correspond to two ORBacus-based IIOP MTP on two different addresses:

```
-mtp
orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cselt.it:1234/jade);orbacus.MessageTransportProtocol(corbaloc:iiop:sharon.cselt.it:5678/jade)
```

When a container starts, it prints on the standard output all the active MTP addresses, separated by a carriage return. Moreover, it writes the same addresses in a file, named:

`MTPs-`<Container Name>`.txt`.

If no MTP related option is given, by default a basic IIOP MTP is activated on the Main Container and no MTP are activated on an ordinary container. To inhibit the creation of the default IIOP endpoint, use the `–nomtp` option.

2.5.2 Configuring MTPs from the graphical management console.

Using the `–mtp` command line option, a transport endpoint lives as long as its container is up; when a container is shut down, all its MTPs are deactivated and the AMS information is updated accordingly. The JADE RMA console enables a more flexible management of the MTPs, allowing activating and deactivating transport protocols during normal platform operations. In the leftmost panel of the RMA GUI, right-clicking on an agent container tree node brings up the popup menu with an *Install a new MTP* and *Uninstall an MTP*.

[1] ORBacus is a CORBA 2.3 ORB for C++ and Java. It is available from Object Oriented Concepts, Inc. at http://www.ooc.com. An alternate IIOP MTP for JADE, exploiting ORBacus features, is available in the download area of the JADE web site: http://jade.cselt.it/.

Choosing *Install a new MTP* a dialog is shown where the user can select the container to install the new MTP on, the fully qualified name of the class implementing the protocol, and (if it is supported by the chosen protocol) the transport address that will be used to contact the new MTP. For example, to install a new IIOP endpoint, using the default JDK 1.3 ORB, one would write `jade.mtp.iiop.MessageTransportProtocol` as the class name and nothing as the address. In order to install a new IIOP endpoint, using the ORBacus based implementation, one would write `orbacus.MessageTransportProtocol` as the class name and (if the endpoint is to be deployed at host `sharon.cselt.it`, on the TCP port 1234, with an object ID `jade`) `corbaloc:iiop:sharon.cselt.it:1234/jade` as the transport address.

Choosing *Uninstall an MTP*, a dialog is shown where the user can select from a list one of the currently installed MTPs and remove it from the platform.

2.5.3 Agent address management

As a consequence of the MTP management described above, during its lifetime a platform, and its agents, can have more than one address and they can be activated and deactivated during the execution of the system. JADE takes care of maintaining consistence within the platform and the addresses in the platform profile, the AMS knowledge base, and in the AID value returned by the method getAID() of the class Agent.

For application-specific purposes, an agent can still decide to choose explicitly a subset of the available addresses to be contacted by the rest of the world. In some cases, the agent could even decide to activate some application specific MTP, that would not belong to the whole platform but only to itself. So, the preferred addresses of an agent are not necessarily the same as the available addresses for its platform. In order to do that, the agent must take care of managing its own copy of agent ID and set the sender of its ACLMessages to its own copy of agent ID rather than the value returned by the method getAID().

2.5.4 Writing new MTPs for JADE

To write a new MTP that can be used by JADE, all that is necessary is to implement a couple of Java interfaces, defined in the `jade.mtp` package. The `MTP` interface models a bi-directional channel that can both send and receive ACL messages (this interface extends the `OutChannel` and `InChannel` interfaces that represent one-way channels). The `TransportAddress` interface is just a simple representation for an URL, allowing separately reading the protocol, host, port and file part.

2.5.4.1 The Basic IIOP MTP

An implementation of the FIPA 2000 IIOP-based transport protocol is included with JADE. This implementation relies on the JDK 1.2 ORB (but can also use the JDK 1.3 ORB, requiring recompilation of the `jade.mtp.iiop` package). This implementation fully supports IOR representations such as `IOR:000000000000001649444c644f4`..., and does not allow to choose the port number or the object key. These limitations are due to the underlying ORB, and can be solved with other JADE MTPs exploiting more advanced CORBA ORBs. The MTP implementation is contained within the `jade.mtp.iiop.MessageTransportProtocol` class, so this is the name to be used when starting the protocol. Due to the limitation stated above, choosing the address explicitly is not supported.

The default IIOP MTP also supports a limited form of `corbaloc:` addressing: A `corbaloc:` address, generated by some other more advanced ORB and pointing to a different platform, can be used to send ACL messages. Interoperability between a JADE platform using ORBacus and a JADE platform using the JDK 1.3 ORB has been successfully tested. In a first test, the first platform exported a `corbaloc:` address generated by ORBacus, and then the second platform used that address with the JDK 1.3 ORB to contact the first one. In a second test, the IOR generated by the second platform was converted into a `corbaloc:` URL via the `getURL()` method call in the `IIOPAddress` inner class (a non-public inner class of the `jade.mtp.iiop.MessageTransportProtocol` class); then the first platform used that address to contact the second one.

So, the `corbaloc:` support is almost complete. The only limitation is that it's not possible to export `corbaloc:` addresses with the JDK 1.3 ORB. JADE is able to convert IORs to `corbaloc:` URLs, but the CORBA object key is an arbitrary octet sequence, so that the resulting URL contains forbidden characters that are escaped using '%' and their hexadecimal value. While this conversion complies with CORBA 2.4 and RFC 2396, the resulting URL is just as unreadable as the plain old IOR. The upcoming JDK 1.4 is stated to feature an ORB that complies with the POA and INS specifications, so that it has persistent object references, and natively supports `corbaloc:` and `corbaname:` addresses. It is likely that a more complete IIOP MTP will be provided for the JDK 1.4, when it will be widely available.

2.5.4.2 The ORBacus MTP

A Message Transport Protocol implementation that complies with FIPA and exploits the ORBacus ORB implementation can be download as an add-on from the JADE web site. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this MTP. This MTP fully supports `IOR:`, `corbaloc:` and `corbaname:` addresses.

According to the OMG specifications, three syntaxes are allowed for an IIOP address (all case-insensitive):

```
IIOPAddress ::= "ior:" (HexDigit HexDigit+)
               | "corbaname://" NSHost ":" NSPort "/" NSObjectID
                "#" objectName
              | "corbaloc:" HostName ":" portNumber "/" objectID
```

Notice that, in the third case, BIG_ENDIAN is assumed by default, while in the first and second case, the endianess information is contained within the IOR definition. In the second form, HostName and PortNumber refer to the host where the CORBA Naming Service is running.

2.5.4.3 The HTTP MTP

A Message Transport Protocol implementation that complies to FIPA and uses the HTTP protocol can be download as an add-on from the JADE web site. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this MTP.

2.6 Support for ACL Codec

By default, all ACLMessages are encoded via the String format defined by FIPA. However, at configuration time it is possible to add additional ACLCodecs that can be used by agents on that container. The command line option *–aclcodec* should be used for this purpose. Agents wishing to send messages with non-default encodings should set the right value in the *aclRepresentation* field of the Envelope.

2.6.1 XML Codec

An XML-based implementation of the ACLCodec can be download from the JADE site as an add-on. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this codec.

2.6.2 Bit Efficient ACL Codec

A bit-efficient implementation of the ACLCodec can be download from the JADE site as an add-on. A tutorial is available in the JADE documentation that describes how to download, install, compile and use this codec. Take care that this codec is available under a different license, not LGPL.

3 AGENT IDENTIFIERS AND SENDING MESSAGES TO REMOTE AGENTS

According to the FIPA specifications, each agent is identified by an Agent Identifier (AID). An Agent Identifier (AID) labels an agent so that it may be distinguished unambiguously within the Agent Universe.

The AID is a structure composed of a number of slots, the most important of which are **name** and **addresses**.

The **name** parameter of an AID is a globally unique identifier that can be used as a unique referring expression of the agent. JADE uses a very simple mechanism to construct this globally unique name by concatenating a user-defined nickname to its home agent platform name (HAP), separated by the '@' character. Therefore, a full valid name in the agent universe, a so-called GUID (Globally Unique Identifier), is peter@kim:1099/JADE where 'peter' is the agent nickname that was specified at the agent creation time, while 'kim:1099/JADE' is the platform name. Only full valid names should be used within ACLMessages.

The **addresses** slot, instead, should contain a number of transport addresses at which the can be contacted. The syntax of these addresses is just a sequence of URI. When using the default IIOP MTP, the URI for all the local addresses is the IOR printed on stdout. The address slot is defaulted to the addresses of the local agent platform.

4 GRAPHICAL USER INTERFACE TO MANAGE AND MONITOR THE AP ACTIVITY

To support the difficult task of debugging multi-agent applications, some tools have been developed. Each tool is packaged as an agent itself, obeying the same rules, the same communication capabilities, and the same life cycle of a generic application agent.

4.1 Remote Monitoring Agent

The Remote Monitoring Agent (RMA) allows controlling the life cycle of the agent platform and of all the registered agents. The distributed architecture of JADE allows also remote controlling, where the GUI is used to control the execution of agents and their life cycle from a remote host.

An RMA is a Java object, instance of the class `jade.tools.rma.rma` and can be launched from the command line as an ordinary agent (i.e. with the command `java jade.Boot myConsole:jade.tools.rma.rma`), or by supplying the '`-gui`' option the command line parameters (i.e. with the command `java jade.Boot –gui`).

More than one RMA can be started on the same platform as long as every instance has a different local name, but only one RMA can be executed on the same agent container.

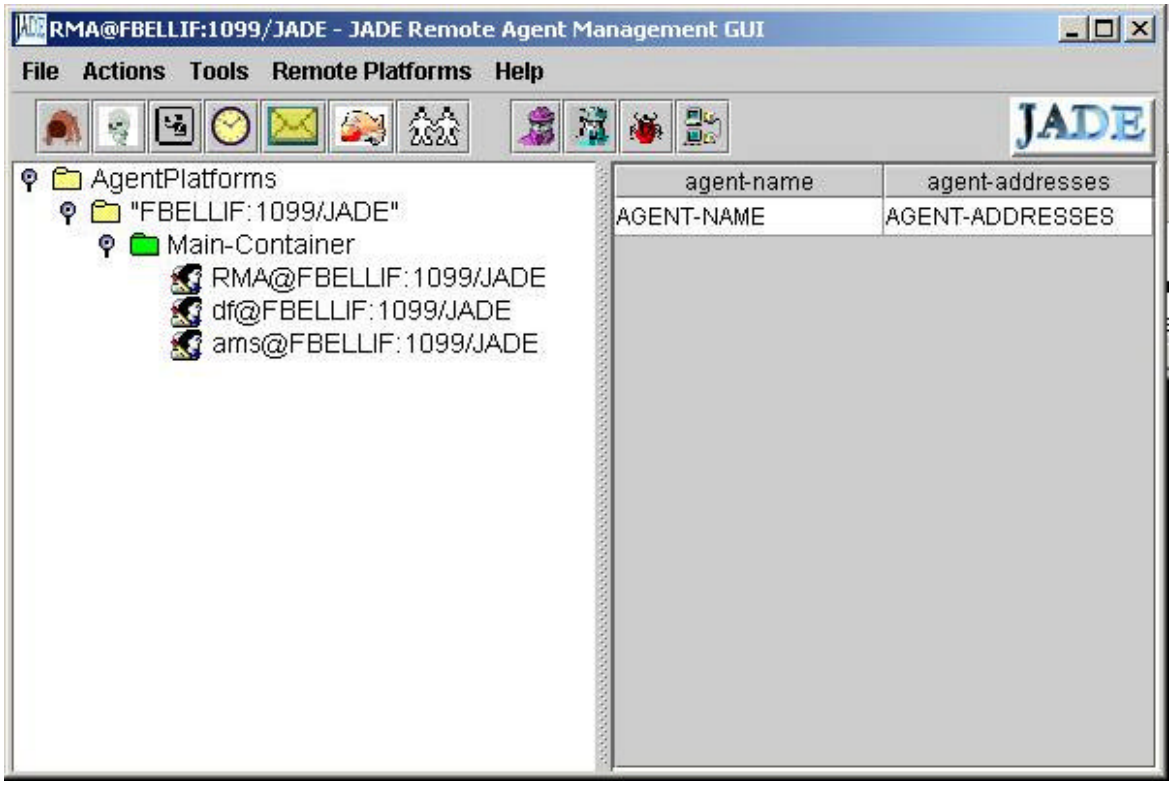


Figure 1 Snapshot of the RMA GUI

The followings are the commands that can be executed from the menu bar (or the tool bar) of the RMA GUI.

- `File menu:`

  This menu contains the general commands to the RMA.

  - `Close RMA Agent`

    Terminates the RMA agent by invoking its `doDelete()` method. The closure of the RMA window has the same effect as invoking this command.

  - `Exit this Container`

    Terminates the agent container where the RMA is living in, by killing the RMA and all the other agents living on that container. If the container is the Agent Platform Main-Container, then the whole platform is shut down.

  - `Shut down Agent Platform`

    Shut down the whole agent platform, terminating all connected containers and all the living agents.

- `Actions menu:`

  This menu contains items to invoke all the various administrative actions needed on the platform as a whole or on a set of agents or agent containers. The requested action is performed by using the current selection of the agent tree as the target; most of these actions are also associated to and can be executed from toolbar buttons.

  - `Start New Agent`

    This action creates a new agent. The user is prompted for the name of the new agent and the name of the Java class the new agent is an instance of. Moreover, if an agent container is currently selected, the agent is created and started on that container; otherwise, the user can write the name of the container he wants the agent to start on. If no container is specified, the agent is launched on the Agent Platform Main-Container.

  - `Kill Selected Items`

    This action kills all the agents and agent containers currently selected. Killing an agent is equivalent to calling its `doDelete()` method, whereas killing an agent container kills all the agents living on the container and then de-registers that container from the platform. Of course, if the Agent Platform Main-Container is currently selected, then the whole platform is shut down.

  - `Suspend Selected Agents`

    This action suspends the selected agents and is equivalent to calling the `doSuspend()` method. Beware that suspending a system agent, particularly the AMS, deadlocks the entire platform.

  - `Resume Selected Agents`

    This action puts the selected agents back into the `AP_ACTIVE` state, provided they were suspended, and works just the same as calling their `doActivate()` method.

  - `Send Custom Message to Selected Agents`

    This action allows to send an ACL message to an agent. When the user selects this menu item, a special dialog is displayed in which an ACL message can be

composed and sent, as shown in the figure.

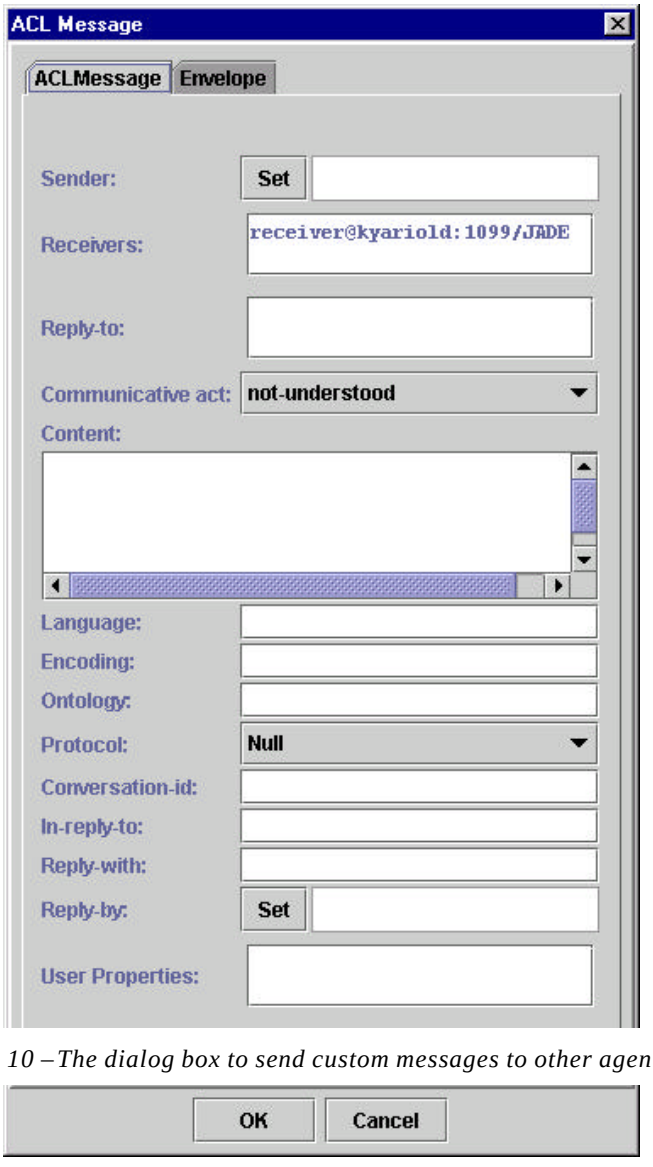


Figure 10 – The dialog box to send custom messages to other agents

- `Migrate Agent`

This action allows to migrate an agent. When the user selects this menu item, a special dialog is displayed in which the user must specify the container of the platform where the selected agent must migrate. Not all the agents can migrate because of lack of serialization support in their implementation. In this case the user can press the cancel button of this dialog.

- `Clone Agent`

  This action allows to clone a selected agent. When the user selects this menu item a dialog is displayed in which the user must write the new name of the agent and the container where the new agent will start.

- `Tools menu:`

  This menu contains the commands to start all the tools provided by JADE to application programmers. These tools will help developing and testing JADE based agent systems.

- `RemotePlatforms menu:`

  This menu allows controlling some remote platforms that comply with the FIPA specifications. Notice that these remote platforms can even be non-JADE platforms.

  - `Add Remote Platform via AMS AID`

    This action allows getting the description (called APDescription in FIPA terminology) of a remote Agent Platform via the remote AMS. The user is requested to insert the AID of the remote AMS and the remote platform is then added to the tree showed in the RMA GUI.

  - `Add Remote Platform via URL`

    This action allows getting the description (called APDescription in FIPA terminology) of a remote Agent Platform via a URL. The content of the URL must be the stringified APDescription, as specified by FIPA. The user is requested to insert the URL that contains the remote APDescription and the remote platform is then added to the tree showed in the RMA GUI.

  - `View APDescription`

    To view the AP Description of a selected platform.

  - `Refresh APDescription`

    This action asks the remote AMS for the APDescription and refresh the old one.

  - `Remove Remote Platform`

    This action permits to remove from the GUI the selected remote platform.

  - `Refresh Agent List`

    This action performs a search with the AMS of the Remote Platform and the full list of agents belonging to the remote platform are then displayed in the tree.

4.2 DummyAgent

The DummyAgent tool allows users to interact with JADE agents in a custom way. The GUI allows composing and sending ACL messages and maintains a list of all ACL messages sent and received. This list can be examined by the user and each message can be viewed in detail or even edited. Furthermore, the message list can be saved to disk and retrieved later. Many instances of the DummyAgent can be started as and where required.

The DummyAgent can both be launched from the Tool menu of the RMA and from the command line, as follows:

```
Java jade.Boot theDummy:jade.tools.DummyAgent.DummyAgent
```

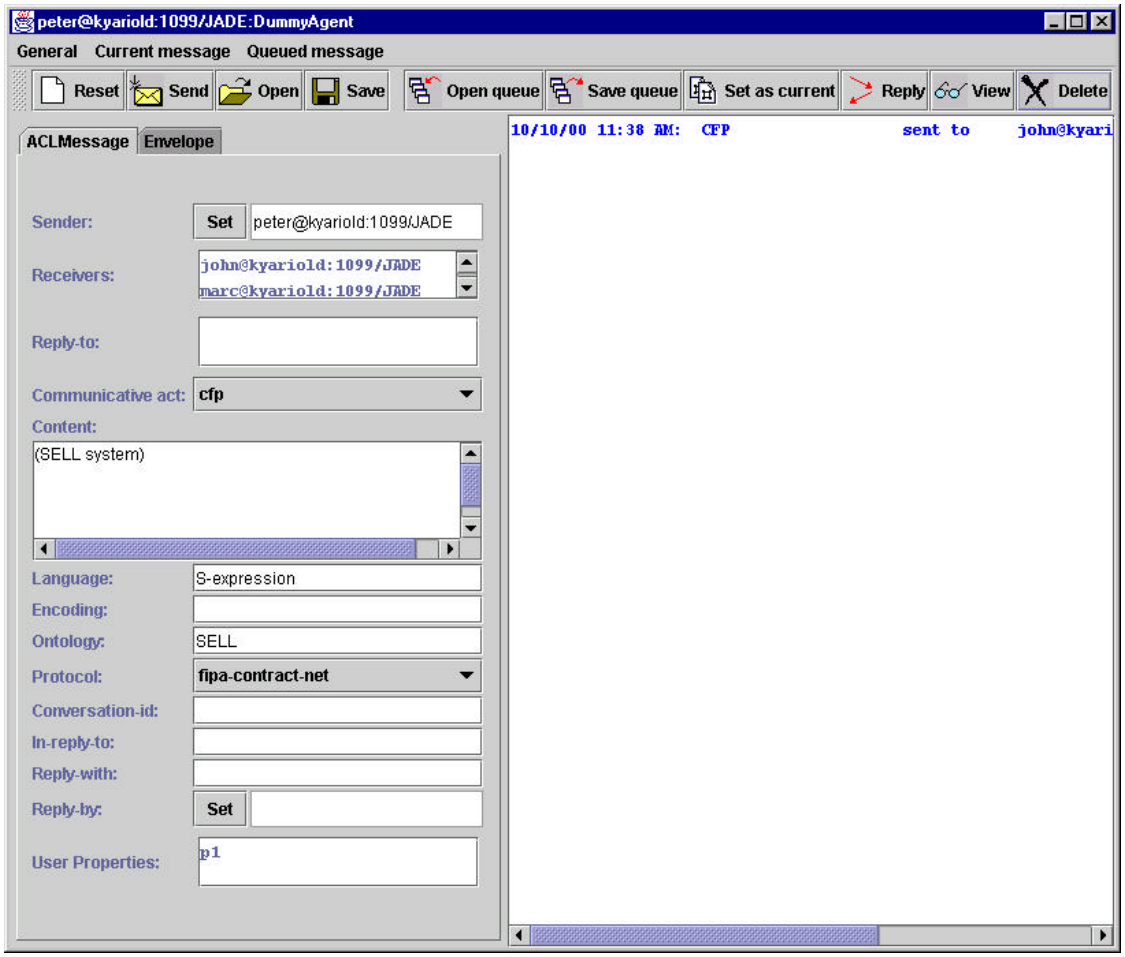


Figure 2 Snapshot of the DummyAgent GUI

4.3 DF GUI

A GUI of the DF can be launched from the Tools menu of the RMA. This action is actually implemented by sending an ACL message to the DF asking it to show its GUI. Therefore, the GUI can just be shown on the host where the platform (main-container) was executed.

By using this GUI, the user can interact with the DF: view the descriptions of the registered agents, register and deregister agents, modify the description of registered agent, and also search for agent descriptions.

The GUI allows also to federate the DF with other DF's and create a complex network of domains and sub-domains of yellow pages. Any federated DF, even if resident on a remote non-JADE agent platform, can also be controlled by the same GUI and the same basic operations (view/register/deregister/modify/search) can be executed on the remote DF.

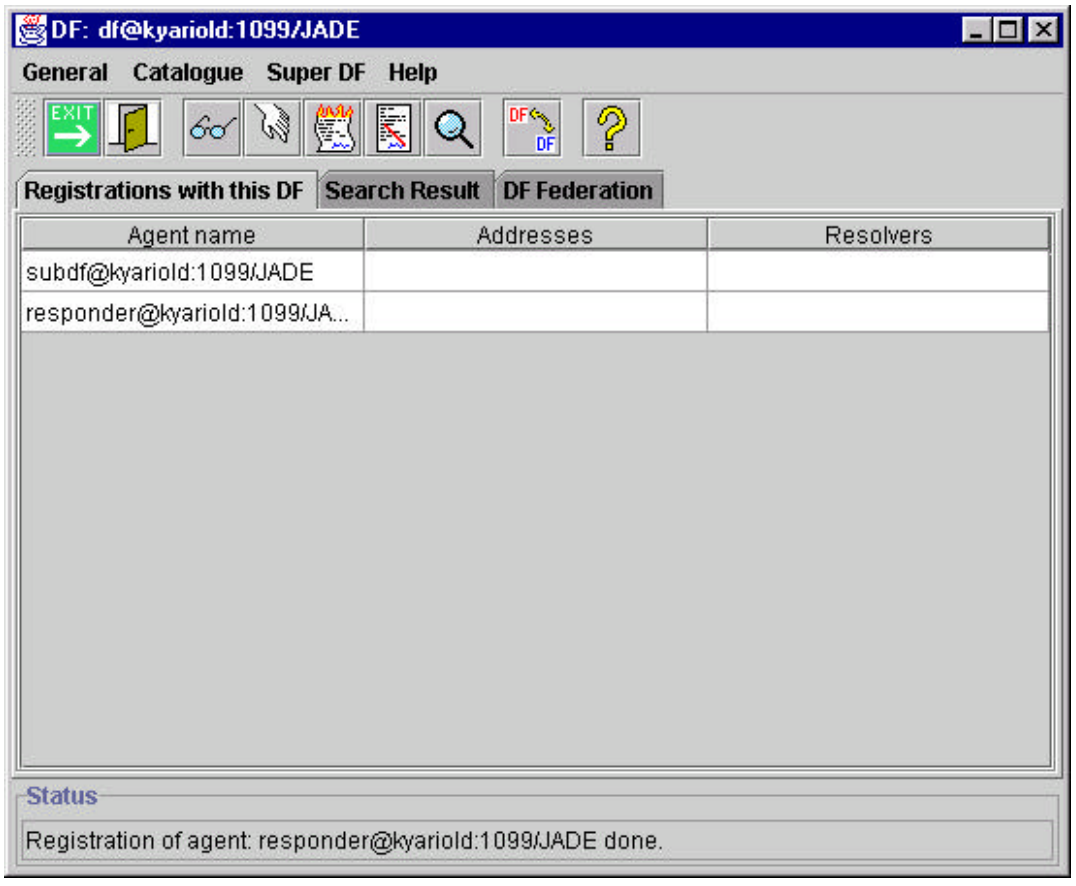


Figure 3 – Snapshot of the GUI of the DF

4.4 Sniffer Agent

As the name itself points out, the Sniffer Agent is basically a Fipa-compliant Agent with sniffing features.

When the user decides to sniff an agent or a group of agents, every message directed to/from that agent / agentgroup is tracked and displayed in the sniffer Gui. The user can view every message and save it to disk. The user can also save all the tracked messages and reload it from a single file for later analysis.

This agent can be started both from the Tools menu of the RMA and also from the command line as follows:

```
java jade.Boot sniffer:jade.tools.sniffer.Sniffer
```

The figure shows a snapshot of the GUI.

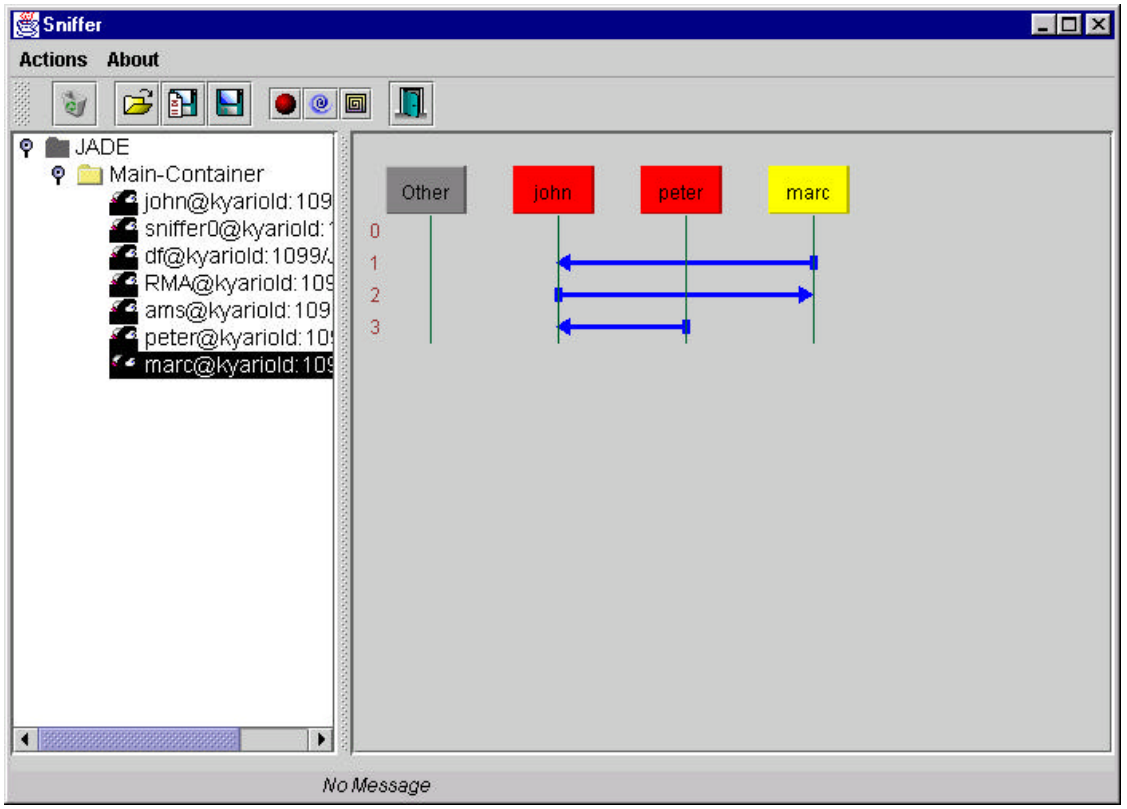


Figure 4 - Snapshot of the sniffer agent GUI

4.5 Introspector Agent

This tool allows to monitor and control the life-cycle of a running agent and its exchanged messages, both the queue of sent and received messages.

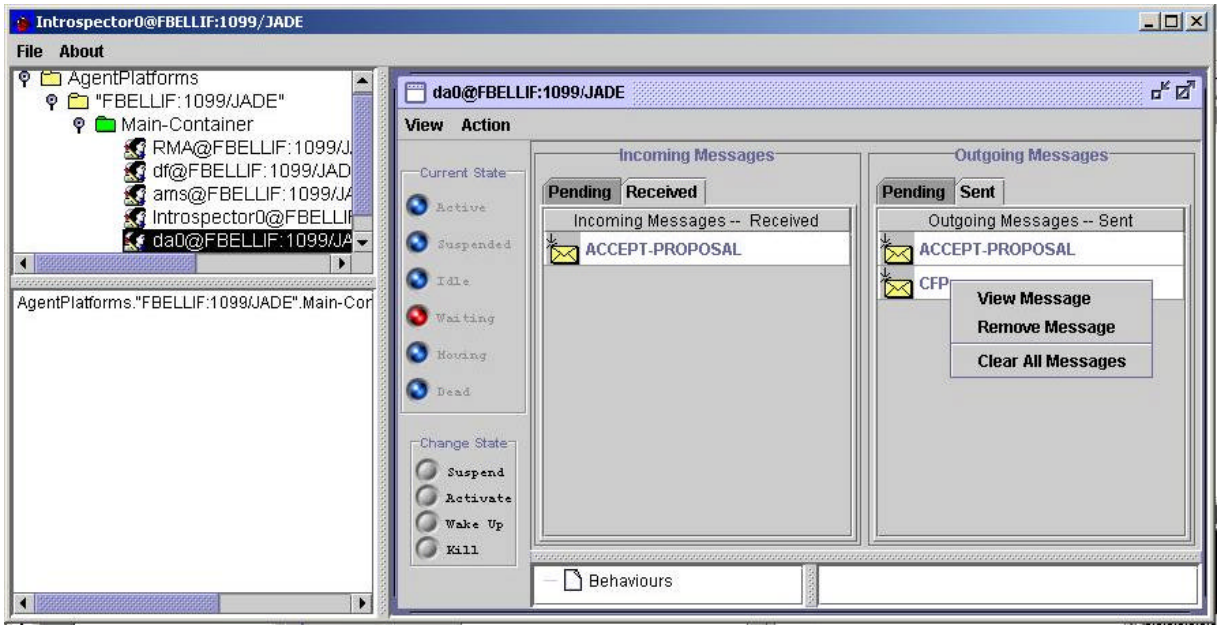


Figure 3 - Snapshot of the Introspector Agent GUI

5 LIST OF ACRONYMS AND ABBREVIATED TERMS

ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management Service. According to the FIPA architecture, this is the agent that is responsible for managing the platform and providing the white-page service.
AP	Agent Platform
API	Application Programming Interface
DF	Directory Facilitator. According to the FIPA architecture, this is the agent that provides the yellow-page service.
EBNF	Extended Backus-Naur Form
FIPA	Foundation for Intelligent Physical Agents
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HAP	Home Agent Platform
HTML	Hyper Text Markup Language
HTTP	Hypertext Transmission Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
INS	
IOR	Interoperable Object Reference
JADE	Java Agent DEvelopment Framework
JDK	Java Development Kit
LGPL	Lesser GNU Public License
MTP	Message Transport Protocol. According to the FIPA architecture, this component is responsible for handling communication with external platforms and agents.
ORB	Object Request Broker
POA	Portable Object Adapter
RMA	Remote Monitoring Agent. In the JADE platform, this type of agent provides a graphical console to monitor and control the platform and, in particular, the life-cycle of its agents.
RMI	Remote Method Invocation
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Tutorial 1: Getting Started with JADE

[revised for JADE 2.5 March 2002]

(Note: The procedures in these notes have been tested on Windows 98 and Windows 2000.)

JADE can be run in several different ways, on one or on many computers. The easiest way is to run a single Jade platform on one computer and use the main container.

Once you have unzipped JADE, you need to make Jade's jar files visible on the classpath. To save typing out could make a one line batch file with the following (on a single line), tailored to your setup,

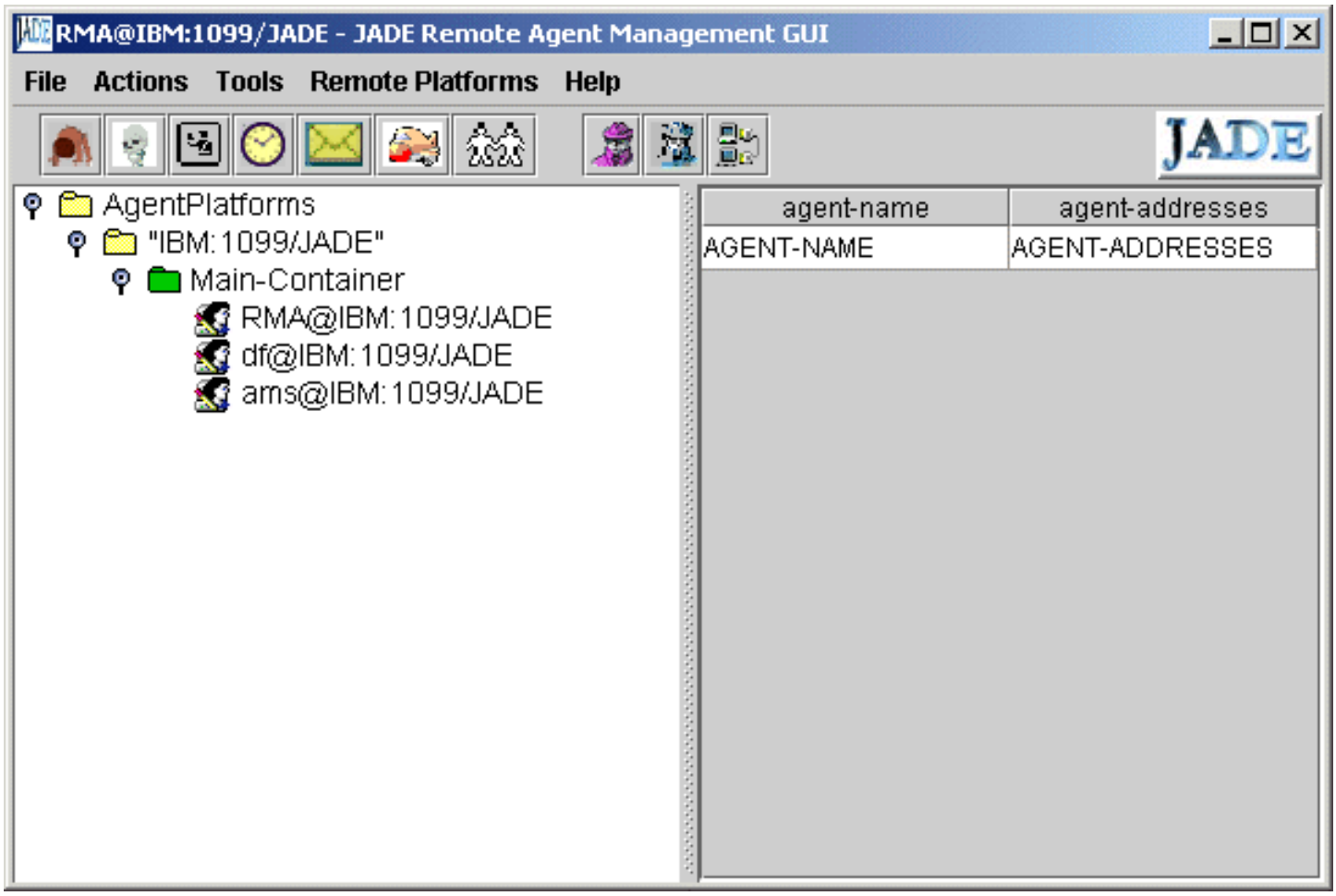
- java -classpath .;.\lib\jade.jar;.\lib\jadeTools.jar;.\lib\iiop.jar;.\lib\base64.jar jade.Boot %1 %2 %3 %4 %5 %6 %7 %8 %9
- We will call this batch file runjade.bat.

Booting Jade

Then boot Jade (from the jade directory),

runjad -gui

You see this window (after you display the tree),



Notes on this image

- Jade agent *platforms* have *containers* to hold agents. A platform can have many containers, not necessarily on the same computer. One container on a platform is "privileged". This *main container* resides on the host which also runs the platform's RMI server. Agents on various containers on a platform use the RMI protocol to communicate.
- The image above shows the GUI of the *Remote Monitoring Agent* (**RMA**) which appears when you use the -gui switch. In addition to itself, the RMA shows the presence of two other agents in the Main Container. The **ams** is the Agent Management System. An agent itself, it provides an environment with many services for agents on the platform. The **df** is the *Directory Facilitator*. It is an agent which provides a "yellow pages" for agents known to the platform.
- Agents must have globally unique names. A name is a "nickname" and an address separated by the at (@) sign. For example, RMA@IBM:1099/JADE is an agent with nickname RMA at the address IBM:1099/JADE. ("IBM" is the name of my Win2000 machine on a LAN.
- The addresses are in RMI format in this case. RMI is used for intra platform communication. (CORBA or HTTP are used for inter platform communication.) The address consists of a host name, in this case IBM, and a port on which the RMI naming service is active, in this case, 1099, the default port for RMI. The name JADE distinguishes Jade RMI invocations from other possible RMI services. Note that in this case, the host name does not have a domain attached. If you wanted a full name you can use the -host switch: java jade.Boot -gui -host jupiter.scs.ryerson.ca, for example. There is also a -port switch if you don't like 1099.

Running Some Agents

We will use the DummyAgent which can be launched by clicking a button on the RMA, and the PingAgent which is an example provided with the Jade distribution. First you need to compile the PingAgent.

Compiling the PingAgent

The source for the PingAgent is in the src\PingAgent directory. I moved it to examples\PingAgent\ under the jade directory. This directory structure matches the package structure declared in the PingAgent.java source file.

You need to compile the PingAgent. I find it convenient to use another one line batch file for compilation (from the Jade directory). The one line might be:

javac -classpath .\lib\jade.jar;.\lib\jadeTools.jar;.\lib\iiop.jar;.\lib\base64.jar;. %1 %2 %3 %4 %5 %6 %7 %8 %9

(all on one line)

I call this file compilejade.bat.

Then compile the Ping Agent with,

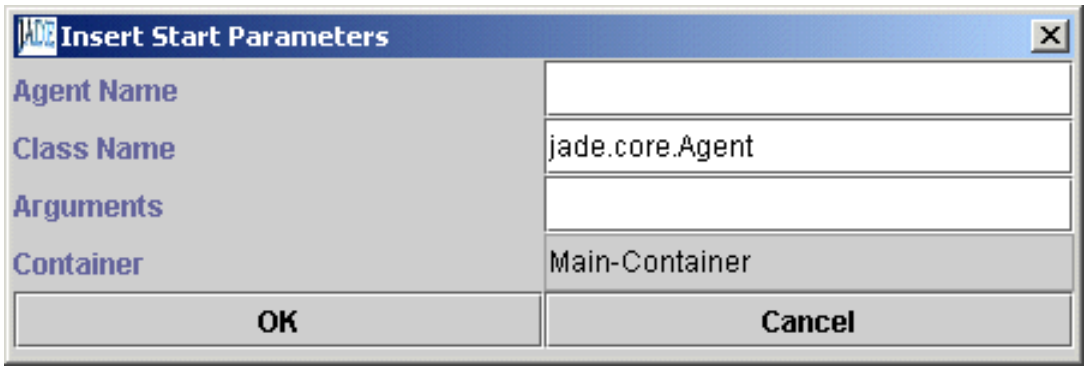
compilejade examples\pingagent\PingAgent.java

Loading the PingAgent into a Jade main container

There are two ways to load agents, using RMA, and from the command line when booting JADE.

Loading agents with the RMA

In the RMA window, select Main-Container, then click the New Agent button (or use the Actions menu). Or you can right click on the Main-Container, and choose Start New Agent. This window pops up:



Enter a name for the agent, say ping0. (In this window just use the nickname of the agent, that is, leave out the address. The address will be filled in by the system.)

Then enter the fully qualified agent class name. In this case, examples.PingAgent.PingAgent. If your class paths are set correctly, after you click OK, the name ping0@IBM:1099/JADE will appear in the Main Container listing. (Of course, the host name will be yours, not mine :-).) If the class cannot be found, JADE will ignore your and may print an error on the Java Console (maybe).

Loading Agents when booting JADE

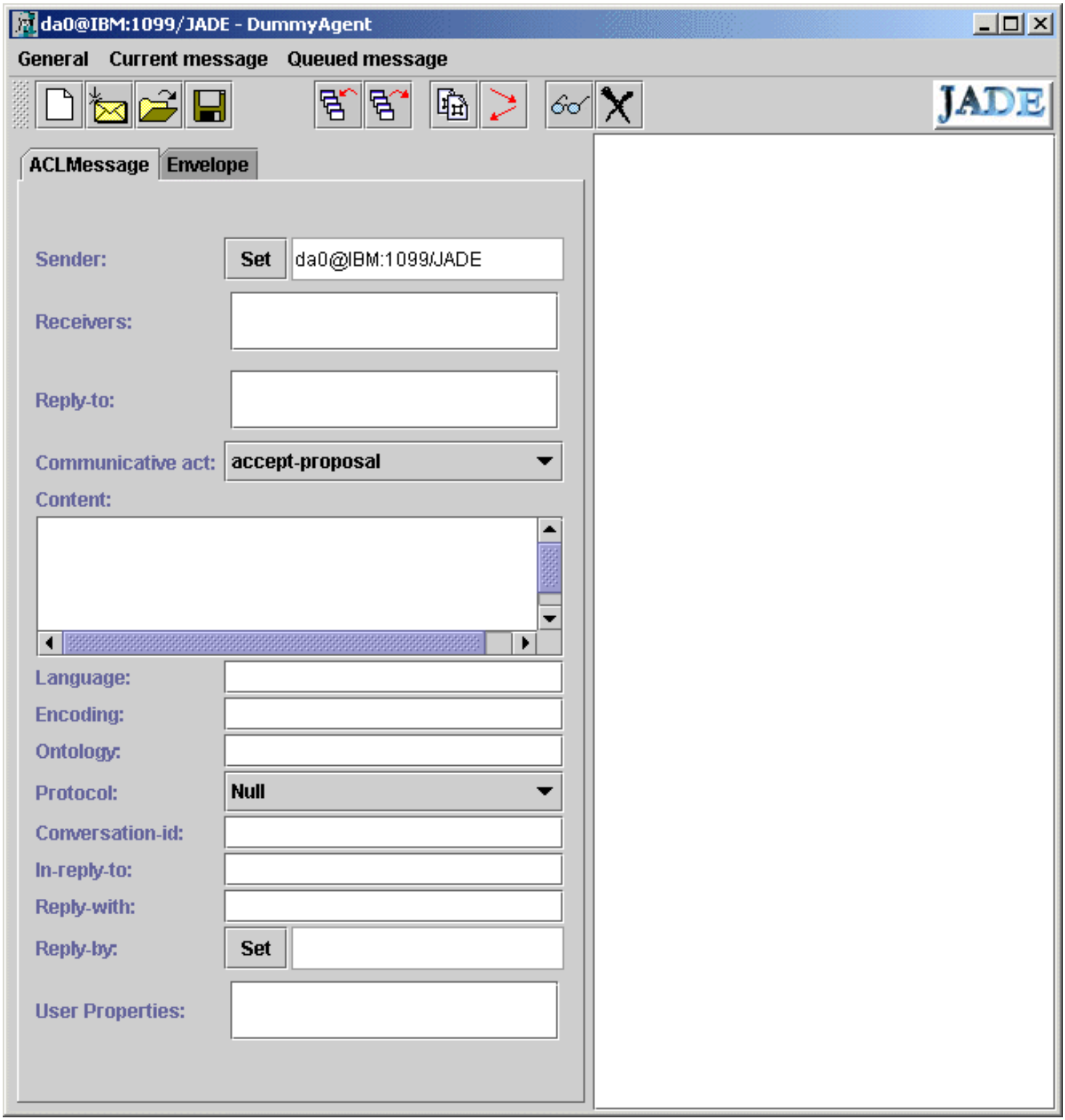
To carry out the same task as above you could have typed,

runjade -gui ping0:examples.PingAgent.PingAgent

and loaded the Ping Agent right away. Note the syntax with the agent nickname separated from its fully qualified class name by a colon.

The Dummy Agent

The Dummy agent has its own button on the RMA. Click it to bring up the DummyAgent window. The Window looks like this:



A formidable form indeed. The form is set up to allow you to describe a Communicative (speech) Act.

Fortunately, at this stage you don't need to know anything about SL. Nor do you have to fill in many fields. The fields you do need to deal with are, receivers, communicative act, and content.

receivers. The receiver is the ping0 agent. With its pointer on the receivers box, *right* click the mouse and select "add". The AID (Agent ID) window appears.

Fill in the form in the manner shown (using your own host name). In the case of the address right click the mouse on the text field.

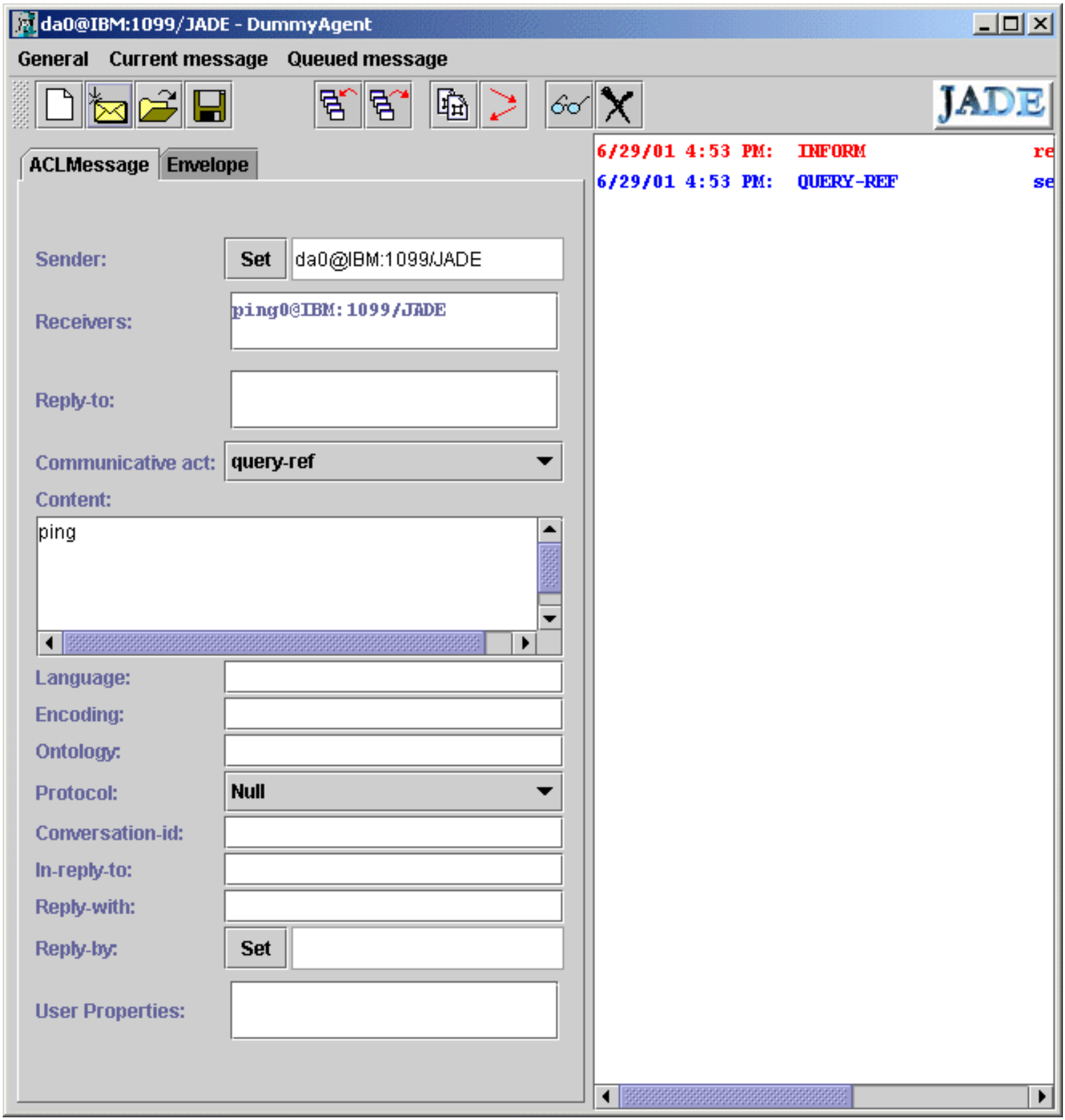
Note the check box. Checking it means the name is local (ping0) in this case. If you don't check it you need to enter the full agent name: ping0@IBM:1099/JADE.

Back in the DummyAgent window, select QUERY-REF for the **communicative act.** In the **request** field, type in the word "ping". (See comment on the PingAgent.java source file.)

Send a message

Finally click the send the message by clicking the send button (second from left).

In the right pane of the DummyAgent window two lines appear, one red, the other blue. The most recent is the topmost. Blue refers to sent messages, red to received messages. You have something like this:



You can examine the received INFORM message (sent by the ping0@IBM:1099/JADE agent by selecting it and then clicking the button with the "glasses" icon.

The ping agent has replied "alive". [In versions of JADE previous to version 2.5, the Ping Agent replies "(pong)".]

Shutting Down the Platform

In the RMA window, choose Shut down platform. Sometimes this does not work. In this case just type ctrl-c in the Java console window to shut down the JVM.

Tutorial 2. JADE Containers, Local and Remote

This tutorial shows how to add local and remote containers to a JADE platform. As before, DummyAgent and PingAgent are used to demonstrate agent communication.

Multiple Containers on One Computer

As in Tutorial 1, open a DOS/Command window,and use runjade.bat as done in Tutorial 1.

runjade -gui -host Frodo

(The local name for the computer is Frodo. You don't really need the -host flag here. But ...)

If you are using computers in different domains you would want to include the full host name to prevent JADE from just using localhost. For example,

runjade -gui -host frodo.scs.ryerson.ca

Display the container tree in the RMA agent window. You see the main container with the df, ams and RMA agents.

Now open another DOS/Command window. In that window we create a satellite container and put the PingAgent in it.

runjade -host Frodo -container ping0:examples.PingAgent.PingAgent

Notes on the syntax

- The -host switch here is used somewhat differently than when used when creating the platform itself (that is, when used with the -gui switch). In the -gui case, -host just names the host where the platform and its RMI naming service are located. *But when you are adding a container to a platform, the -host switch tells which host is hosting the platform you want to hook up your container to.*
- The -container switch tells the system that this is just a container. After this switch you can list agents which you want to put in the container (separated by spaces). If your agent needs command line arguments you can list them inside parentheses immediately following the agent's name.
- You specify an agent by its nickname, followed by a colon, followed by its fully qualified class name.

Try out the agents

Display the RMA window you will see an entry "container-1" added. Expanding the tree shows the ping0 agent.

Invoke the DummyAgent and use it to send a message to ping0 in the same way as was done in Tutorial

1. (Fill in the receivers=ping0@Frodo:1099/JADE, communicative act=QUERY-REF, Content=ping) If you examine the return message it is the same as in Tutorial 1, namely "(pong)" or "alive", depending on your JADE version..

Try changing the commutative act to INFORM and send the message to ping0. You will get a NOT-UNDERSTOOD reply. Look at the content of this message for an example of the SL language constructed by JADE.

Remote Containers

So far, this tutorial is just a repeat of Tutorial 1. It is not surprising to find that you can send messages from agent to agent on the same platform whether they are in different containers or not. More interesting is that JADE is a distributed system. A platform can have containers on remote systems as well as locally. So if you have a second computer networked to the first, try this.

On the second computer set up and run a JADE container with a PingAgent in it. In other words, simply, type,

runjade -host Frodo -container ping1:examples.PingAgent.PingAgent.

This is exactly the same command as used above to create a container on machine Frodo itself! Thanks to RMI, the system is transparent with respect to hosts. Notice that, since an agent nicknamed ping0 already exists on this platform, I must use a different nickname for this second PingAgent, even though the new agent is running on a different machine.

Looking on the original machine (Frodo in my case) I see that a new container, Container-2 has appeared on the RMA agent window in which ping1 is listed. I can, once again, send the usual message from the DummyAgent on Frodo to ping1 which lives on another machine (named IBM in my case). The location of ping1 is transparent to the user of RMA on Frodo. It could be anywhere.

The container on the other computer is a client of the RMI server running on Frodo. So is ping0 a client which happens to exist on the same machine as the RMI server itself.

Running an RMA (Remote Monitoring Agent) with a container

If you have followed the tutorial to this point you have a main JADE platform on one machine (Frodo in my case). This machine is the home of the Main Container and the RMI server. A second container, Container-1 is installed on this machine and contains one agent, ping0@Frodo:1099/JADE, to give it its full name. On a second machine, IBM in my case, there is a second container, Container-2 containing one agent, ping1@Frodo:1099/JADE. This agent lives on machine IBM but "belongs" to the platform running on machine Frodo.

A user on machine IBM can't see anything or do anything. The user of machine Frodo is in complete control. It would be nice to run an RMA agent and a DummyAgent on a machine remote from the platform machine.

To do this we need to know the fully qualified names of Dummy agent and RMA agent. These are,

- java.tools.DummyAgent.DummyAgent

- java.tools.rma.rma

So start another container with these two agents and yet another ping agent, ping2, on some machine connected to the JADE platform machine. For example,

> runjade -host Frodo -container dummy0:jade.tools.DummyAgent.DummyAgent RMA1:jade.tools.rma.rma

(Don't forget to include a nickname for your agents! AND make sure the names don't clash with names on other containers on the platform. Note RMA1 not RMA.)

You will get the windows for the two agents on the remote machine. RMA1 will show all the containers and agents on the platform, just like RMA on the main platform server. Try sending one of the ping agents the usual message from dummy0.

Note

- If you just launch the RMA agent on the remote machine and try using its button to launch the Dummy Agent, the Dummy Agent fails to appear if one is running elsewhere. Instead, select the container, right click and choose Start New Agent. Choose a none-clashing nickname for the new Dummy Agent, and enter its full class name jade.tools.DummyAgent.DummyAgent.
- A JADE platform is a cooperative system. If you hook up to the main platform server from a remote machine and run an RMA agent on the remote machine you can use that agent to destroy the whole platform! Be nice. Security will eventually be added to JADE.

Tutorial 3. JADE Inter Platform Communication

In Tutorial 2 we saw a simple JADE platform consisting of containers which could be scattered among many different hosts with one Main container on a host running the RMI service. All the agents on one platform communicate using the RMI protocol. RMI is the intra communication mechanism internal to a platform.

JADE agents can also communicate among separate platforms, in other words, JADE provides mechanisms for inter platform communication. These mechanisms are CORBA based and use the IIOP protocol rather than RMI. Such an arrangement allows JADE agents to talk to FIPA compliant agents on any platform supporting CORBA and FIPA standards. This is the whole idea of FIPA: to make diverse agent systems interoperable.

Also, there exist other protocols for inter platform communication. One popular method uses the HTTP protocol.

This tutorial shows how to use the basic Sun IIOP for inter platform communication.

In this tutorial you just get the DummyAgent talking to the PingAgent, one on one platform, the other on another platform. You could run two platforms on one machine, provided you use different ports for the RMI servers. But things are clearer and more typical if the platforms run on different hosts.

What is that big number that appears when booting a platform?

When you invoke java jade.Boot -gui, you get something like this on the Java console (DOS window):

IOR:000000000000001149444C3A464950412F4D54533A312E3000000000000000010000000000000540001010000000000C3139322E3136382E302E350007CE000000000018AFABCAFF0000000223429867000000080000000000000000000001000000010000001400000000000100200000000000010100000000000

You will also find it stored in a file called MTP-Main-Container.txt in the *jade* directory.

What is it? Well, it acts as a CORBA address, or end point. This strange value occurs because by default JADE uses Sun's JDK 1.3 CORBA/iiop implementation which is rather primitive. (The JADE team has provided a more sophisticated ORB which allows explicit addressing.)

To send messages between default versions of JADE platforms you need to use this end point as an agent address, which is rather awkward.

Setting Up

Boot a JADE platform on each of two computers in the usual way. For example, on host Frodo

runjade -host Frodo -gui

and

runjade -host IBM -gui

on host IBM.

Now, as usual, we want to send a message from a DummyAgent to a PingAgent, this time with the agents residing on separate platforms, not just in separate containers on the same platform. To do this we use the IOR's as the addresses for the agents.

End Points

The IIOP IOR (endpoint) of a platform is automatically stored in a file called MTP-Main-Containrer.txt. A sender of a message must know the receivers IOR so somehow, the sender must get a hold of the MTP-Main-Container.txt file of the receiver. One could use ftp. On a LAN with MS Windows you can just copy the file or even just cut and paste its contents.

The platform with the DummyAgent needs to know the IOR of the platform with the PingAgent in order to send it a message. (Of course it knows its own IOR.)

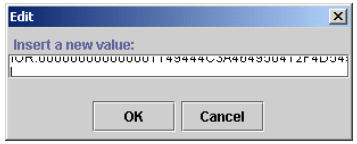
In this tutorial we just use "Network Neighborhood" with the editor NotePad to open the receiver's file and copy the IOR to the Receiver field of the DummyAgent.
On MS Windows you type ctrl-v to do the pasting.

Remember to remove the cr/lf from the end of the IOR. You can see from the pictures below that you get a visual clue if you have entered the IOR correctly.

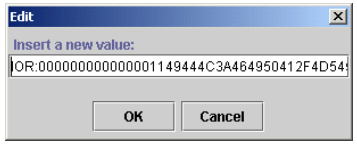
Recall that to add a receiver to the DummyAgent right click on the receivers field and choose add. This brings up the AID window. Enter the full name of the PingAgent, e.g., ping0@Frodo:1099/JADE. Then right click the address field in the AID window and use ctrl-v to paste the IOR into this field.

The IOR with a cr/lf (bad)

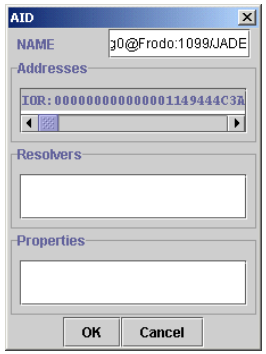
(This may not happen in version 2.5.)



The IOR correctly entered without a CR/LF



The AID window should look something like this:



Do not check the box indicating a local name. Fill in the full,global name (with the @ sign.)

Click OK and then send the message to the ping0 agent.

Send the Message

Finally, you are ready to send the same message as before, from the DummyAgent to the PingAgent. Everything should run just like in all the previous tutorial examples. The QUERY-REF informative is sent by the DummyAgent, and the PingAgent sends an INFORM message in reply. Click the "eyeglasses" button to see the content of the reply.

Note that the message sent by the DummyAgent includes the return address (IOR of the platform on which the DummyAgent is running so you only need to copy one IOR, that of the receiver.

Comment

Using the JDK 1.3 CORBA/IIOP ORB is clearly very awkward. It would be well to install the Orbacus ORB add on (see the distribution documentation on doing this) which allows the standard host:port addressing for the endpoints.

Many people use the HTTP MTP add-on for inter platform communication. Tutorial 4 of these notes shows how to do this.

Contacting the Remote Platform

You will notice a menu entry of the RMA agent (the GUI for the platform) called "Remote Platforms". If you click it you see two ways. The easiest is "add platform via AMS AID.

If you click this you get a window which allows you to enter the agent name and location. For the name you need the global name of the target AMS, for example, ams@frodo:1099/JADE. For the location you need to copy in the IOR of that platform.

You should see a new entry in the RMA agent for the remote platform. To view its contents select and right click. The choose "Refresh agent list" and you should see all the agents on the remote platform.

Tutorial 4: Using the HTTP MTP for Inter Platform Communication

An alternative for inter platform communication using the HTTP protocol instead of IIOP (see Tutorial 3) has been provided with the JADE distribution. It is easy to use and provides a well known and universally used message transport protocol (MTP).

Compiling (if necessary)

The HTTP MTP add-on comes with source and a jar file (htttp.jar) containing the compiled classes. The compiled code assumes you ar using the Crimson SAX parser (crimson.jar from Apache - a free download). If you want to use a different parser, e.g. Xerces, you need to follow the provided instructions. What you want to create is the file http.jar.

Using the HTTP MTP

Once you have http.jar and crimson.jar the rest is easy. I just put these in the lib directory along with the other JADE jar files in the lib directory. Then I just needed to tell JADE where the main class is using the -mtp command line switch. All this can be put in a one line batch file which looks like this:

java -classpath
.;.\lib\jade.jar;.\lib\jadeTools.jar;.\lib\iiop.jar;.\lib\base64.jar;.\lib\crimson.jar;.\lib\http.jar
jade.Boot -mtp jamr.jademtp.http.MessageTransportProtocol %1 %2 %3 %4 %5 %6 %7 %8 %9

You could call this runjadehttp.bat.

The Ping Agent Example (again)

The easiest way to run is to load both the HTTP MTP and the Ping Agent from the command line like so:

runjadehttp -gui ping0:examples.PingAgent.PingAgent

Now you will not see the huge IOR number for the end point of this platform but rather something more human:

> **This is JADE 2.5 - 2002/02/05 14:01:24**
> **downloaded in Open Source, under LGPL restrictions,**
> **at http://jade.cselt.it/**
>
> **http://IBM:7778/acc**
> **Agent container Main-Container@JADE-IMTP://IBM is ready.**

http://IBM:7778/acc is the address of the platform on the host "IBM" on the default port 7778. You can override the defaults with -port and -host as usual.

Similarly run another platform on another host. In my case this is a computer called Frodo, so the platform has the address http://Frodo:7778/acc.

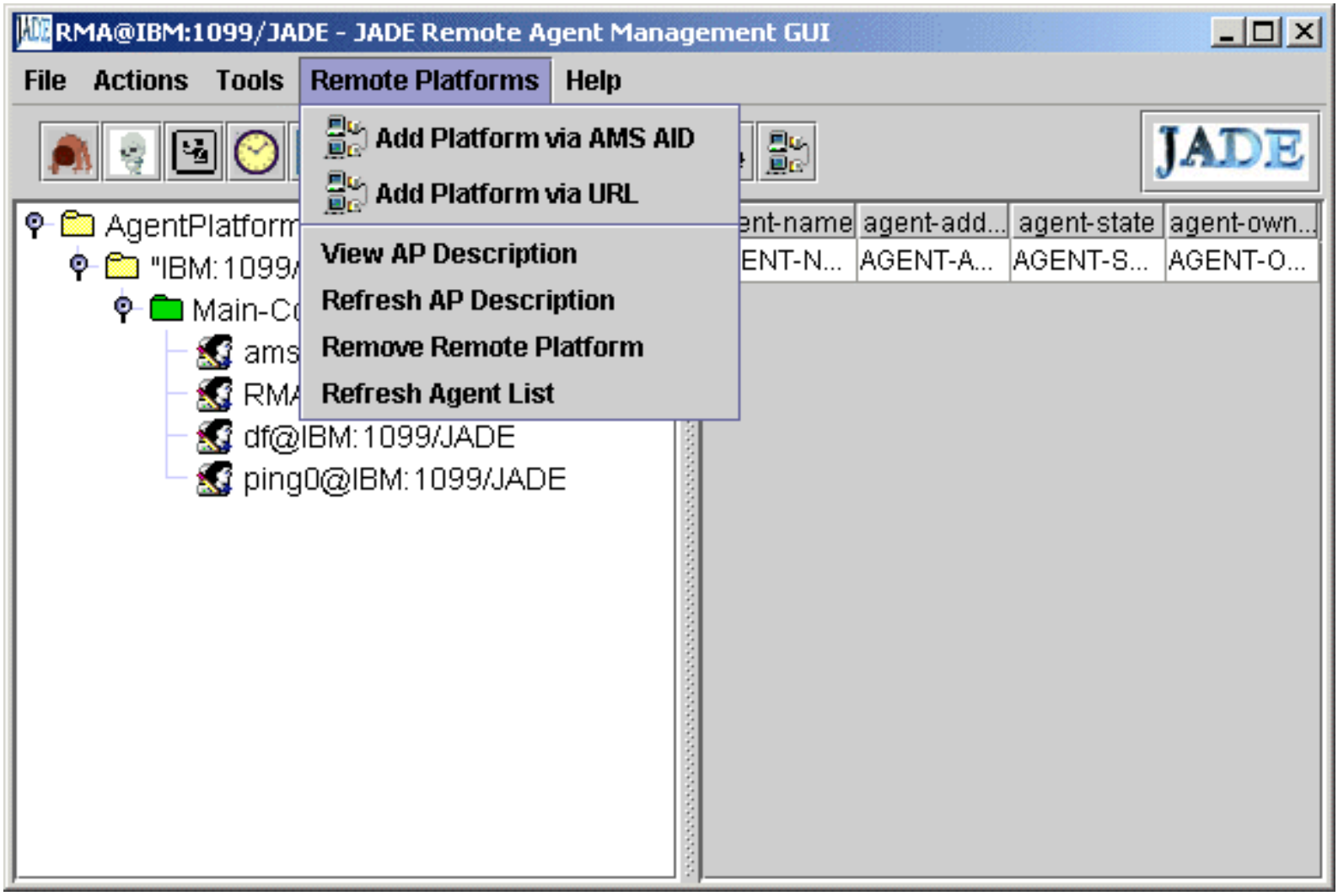
(Reminder: If you need the full name of the host, including the domain, specify it with -host, e.g. Frodo.scss.ryerson.ca, on the command line.)

Sending a Message to the Remote Platform

Suppose we are on host IBM and want to send the "ping" message to a Ping Agent on host Frodo. But is there a Ping Agent there? And, if so, what is its name? We need some info from the remote platform. This is where the Remote Manager Agent (RMA) comes in.

Contacting a Remote Host

On the RMA menu select "remote platforms". You see this.

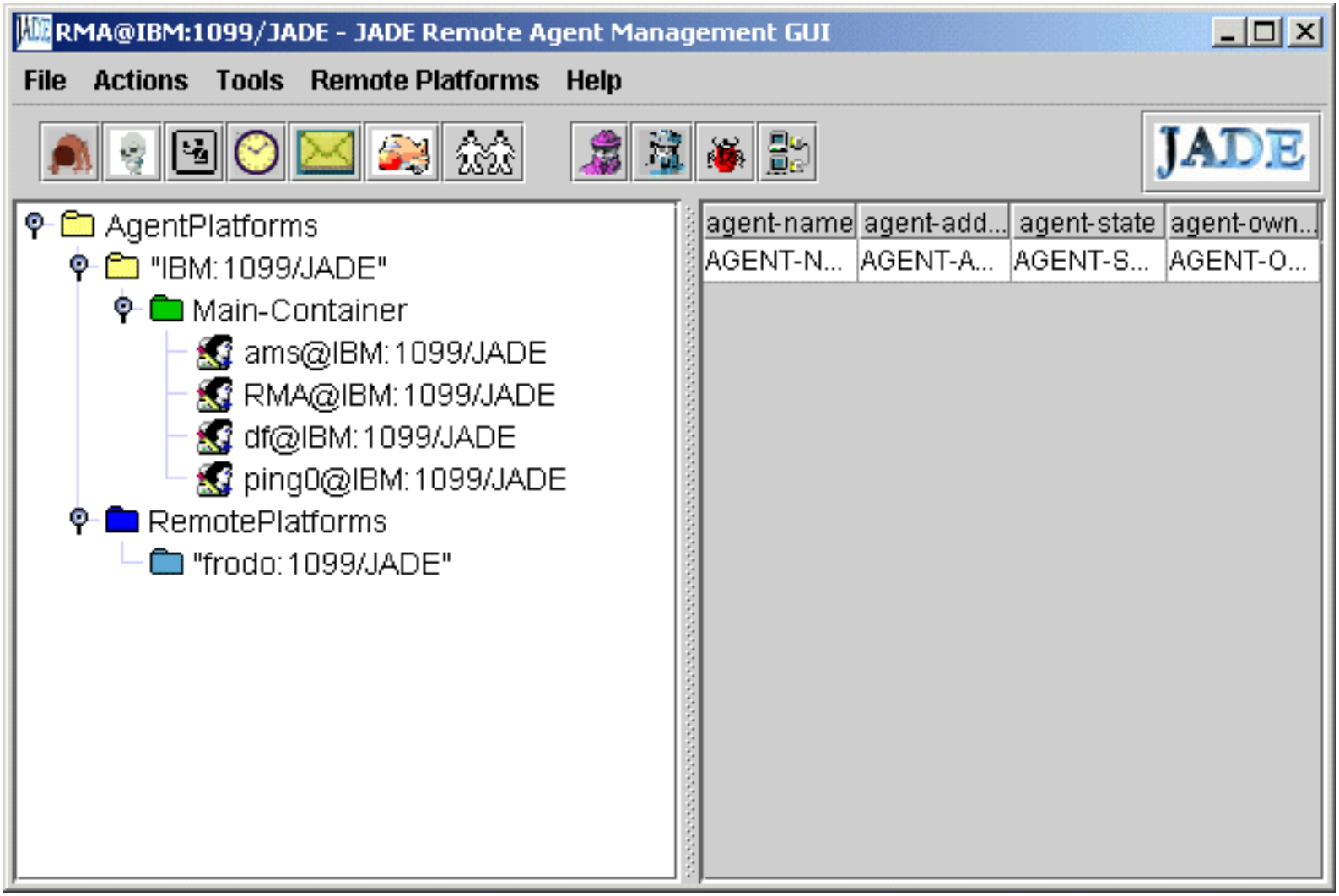


Select "Add Platform via AMS AID". This window appears. (Shown filled out>)

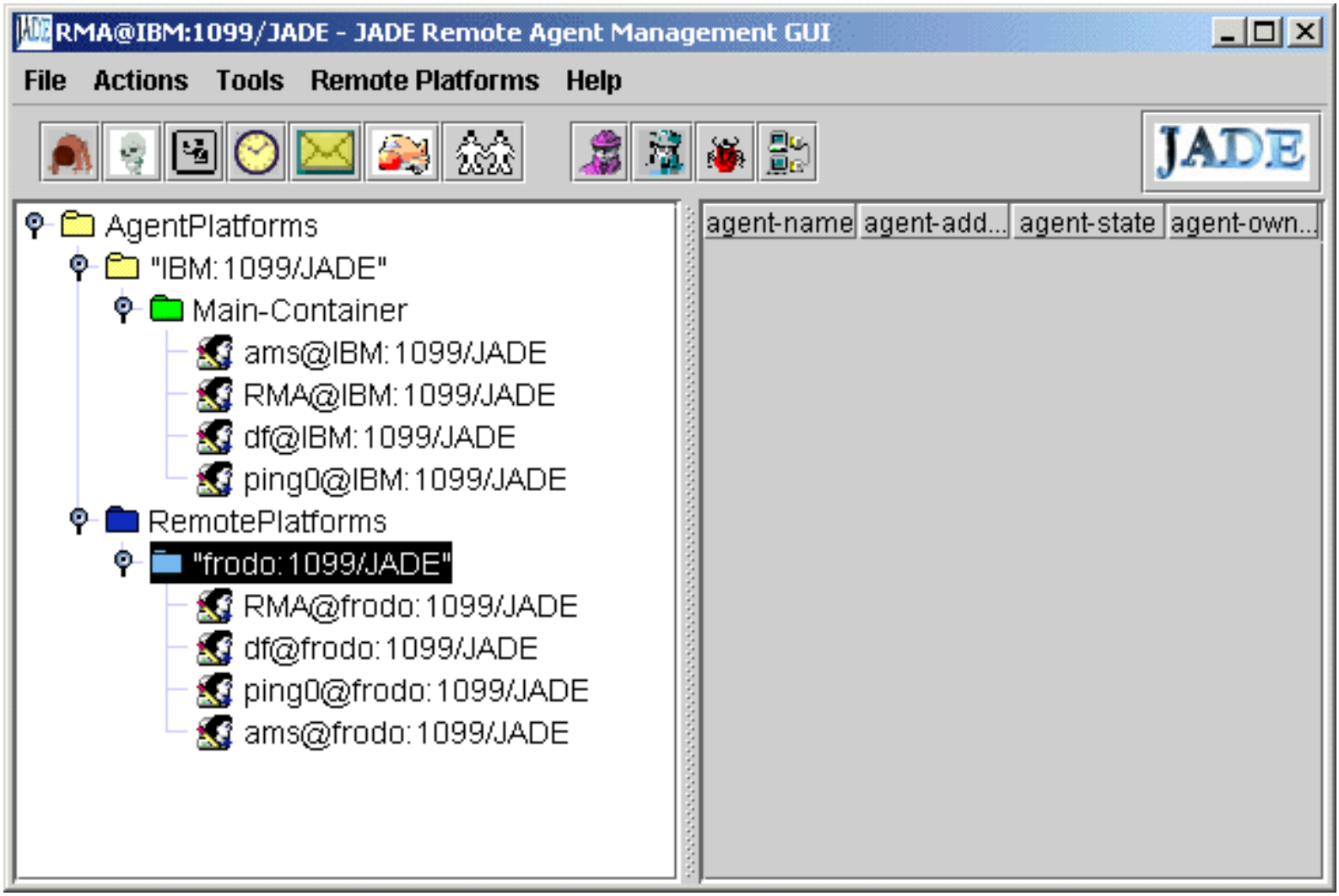
Notice the name of the ams agent on the remote platform. The host part of the name is in RMI format. Note also that the checkbox is unselected. We need the Global Agent Identifier for the remote AMS.

On the other hand the address of the platform is HTTP since we added the HTTP MTP for inter platform communication.

After you click OK, and expand the platform you get this:



Nice, but where are the agents? To see them, select "frodo:1099/JADE" (actually, your equivalent to this), and right click the mouse. Then choose "refresh agent list", and there they are, like so:



Finally, Actually send a message!

Start up the Dummy Agent. Make sure the Communicative Act is QUERY-REF, and the Content is ping.

For the receiver name enter the global name of the Ping Agent on the remote platform, ping0@frodo:1099/JADE in the example. For the address (Right click on the Address field, remember?) enter the HTTP address: http://Frodo:7778/acc. Then send the message.

You should get an INFORM message back which you can inspect by clicking the "glasses" button. You should see "alive". (If you are running an older version of Ping Agent, you may see "(pong)". The change was made in order to conform to an Agent Cities test suite.)

Adding an MTP with the RMA

If you do not use the -mtp command line switch, the Sun IIOP is added by default. You can add other MTPs (or remove them) once the RMA is running. Select the main container and right click, choosing, Add MTP.

To add the HTTP MTP, fill in the field for class name with jamr.jademtp.http.MessageTransportProtocol. Make sure http.jar and crimson.jar are on your class path before booting JADE.

How to use the HTTP MTP with JADE

Author: Ion Constantinescu (EPFL)

Date: May 31, 2001

Java platform: Sun JDK 1.2 Windows

JADE version 2.1

Since JADE 2.1, FIPA-compliant Message Transport Protocols can be plugged and activated at run-time on any JADE container. By default, the platform uses an IIOP based MTP which relies on the ORB provided with jdk1.2. However, HTTP can be used as an alternative transport. This tutorial describes how to install and use the HTTP MTP with JADE.

Installation.

In order to install HTTP the following steps must be performed:

- The HTTP MTP must be downloaded from the JADE download page.
- after downloading you **MUST** unzip the HTTP MTP package under the root of the jade distribution tree. You should end having a hierarchy like jade/add-ons/http.
- A SAX parser must be downloaded and installed into the system. See below a list of known parsers and configuration options.
- The xml parser jar file must be added to the CLASSPATH or specified in the -classpath argument when starting the virtual machine

Compiling

The default Makefile rules don't take the HTTP MTP into account. For handling the compilation process of the HTTP MTP you have to use the Makefile located in the http directory. The following rules are available:

- make - compiles the http classes
- make lib - creates the http.jar archive in the lib directory
- make clean - removes the compiled classes and the http.jar archive
- make batch - creates batch files equivalent to make rules:
  - make.bat - on windows same as make
  - makelib.bat - on windows same as make lib
  - makeclean.bat - on windows same as make clean

Configuration and Usage

The current implementation has been tested with the following parsers:

Parser Name	Parser Class
Crimson	org.apache.crimson.parser.XMLReaderImpl
Xerces	org.apache.xerces.parsers.SAXParser

The current configuration uses Crimson as the default parser. So if you don't want to make any changes you just have to download Crimson from the link provided above and make sure it is added to the classpath when starting (either by including it into the $CLASSPATH environment variable - %CLASSPATH% under windows or by specifing it on the command line) .

Here is an example of how you would start the platform assuming that you copied crimson.jar from the initial distribution to the jade/lib directory:

```
java -classpath
./lib/jade.jar:./lib/jadeTools.jar:./lib/crimson.jar:./http/lib/http.jar
jade.Boot
```
(for Unix)
or
```
java -classpath
.\lib\jade.jar;.\lib\jadeTools.jar;.\lib\crimson.jar;.\http\lib\http.jar
jade.Boot
```
(for Windows)

If you want to use another parser suplementary you have to specify in the command line the system property *org.xml.sax.parser* as in the following example (also assuming that you have copied xerces.jar from the initial distribution to the jade/lib directory) :

```
java -Dorg.xml.sax.parser=org.apache.xerces.parsers.SAXParser -classpath
./lib/jade.jar:./lib/jadeTools.jar:./lib/xerces.jar:./http/lib/http.jar
jade.Boot
```
(for Unix)
or
```
java -Dorg.xml.sax.parser=org.apache.xerces.parsers.SAXParser -classpath
.\lib\jade.jar;.\lib\jadeTools.jar;.\lib\xerces.jar;.\http\lib\http.jar
jade.Boot
```
(for Windows)

It is possible to activate one ore more communication endpoints. There are two main ways for doing such an activation:

- from the command line when you start a JADE container.
- from the GUI of the RMA

Configuring MTPs from the command line.

In this case the following parameter must be specified:

- -mtp jamr.jademtp.http.MessageTransportProtocol -which will start the MTP on the default address port

- -mtp 'jamr.jademtp.http.MessageTransportProtocol(http://myaddress.com:9999/acc)' - which will start the MTP on the port / host given as address. Note that in case of some platforms /configurations the ' might be required.

Configuring MTPs from the graphical management console.

Select a container from the GUI, click the right button of the mouse and a popup menu appears. Choose the Install a new MTP option and a dialog will be shown. Here the following information can be set:

- the container to install the new MTP on (if different from the selected one)
- the fully qualified name of the class implementing the jade.mtp.MTP interface, and (if it is supported by the chosen protocol)
- optionally the transport address that will be used to contact the new MTP.

For example, in order to install a new HTTP endpoint on the default local port by using the HTTP MTP, one should write jamr.jademtp.http.MessageTransportProtocol as the class name and nothing as the address. In order to use the transport on a different port or a particular interface of the current machine you could provide as the transport address a standard http url: http://mymachinename.org:8978 (where 8978 would be the port number on which the transport will bind).

Choosing Uninstall an MTP shows a dialog where the user can select from a list one of the currently installed MTPs and remove it from the platform.

Notes:

When activated the HTTP MTP uses by default the local port 7778. Please take into consideration that using other dedicated ports (such as 80, 8080, etc. on a machine running a web server or a proxy server) might result in configuration conflicts and unpredictable results.

When activating the HTTP MTP from the command line it is preferable to specify also the full transport address - preventing the binding of the server socket to addresses not accessible from outside the domain.

FIPA and JADE

JADE is designed to conform with the FIPA Agent model in order to be able to inter-operate with other FIPA compliant systems such as Zeus (British Telcom) and FIPA-OS.

The FIPA agent specifications

FIPA Agent Management Specifications

There are several specifications most useful to understanding the JADE platform. These are,

The Agent Management Services (AMS) specification XC00023

This includes the Directory Facilitator (DF), naming conventions, the Agent Management System, etc.

The ACL Message Structure Specification XC00061

The Communicative Act Library XC00037

The Conent Language Specification XC0007, 0008,000 9, 0010, 0011.

Of these, SL (XC0008).

In addition there are many other specifications,for example, for message transport protocols.

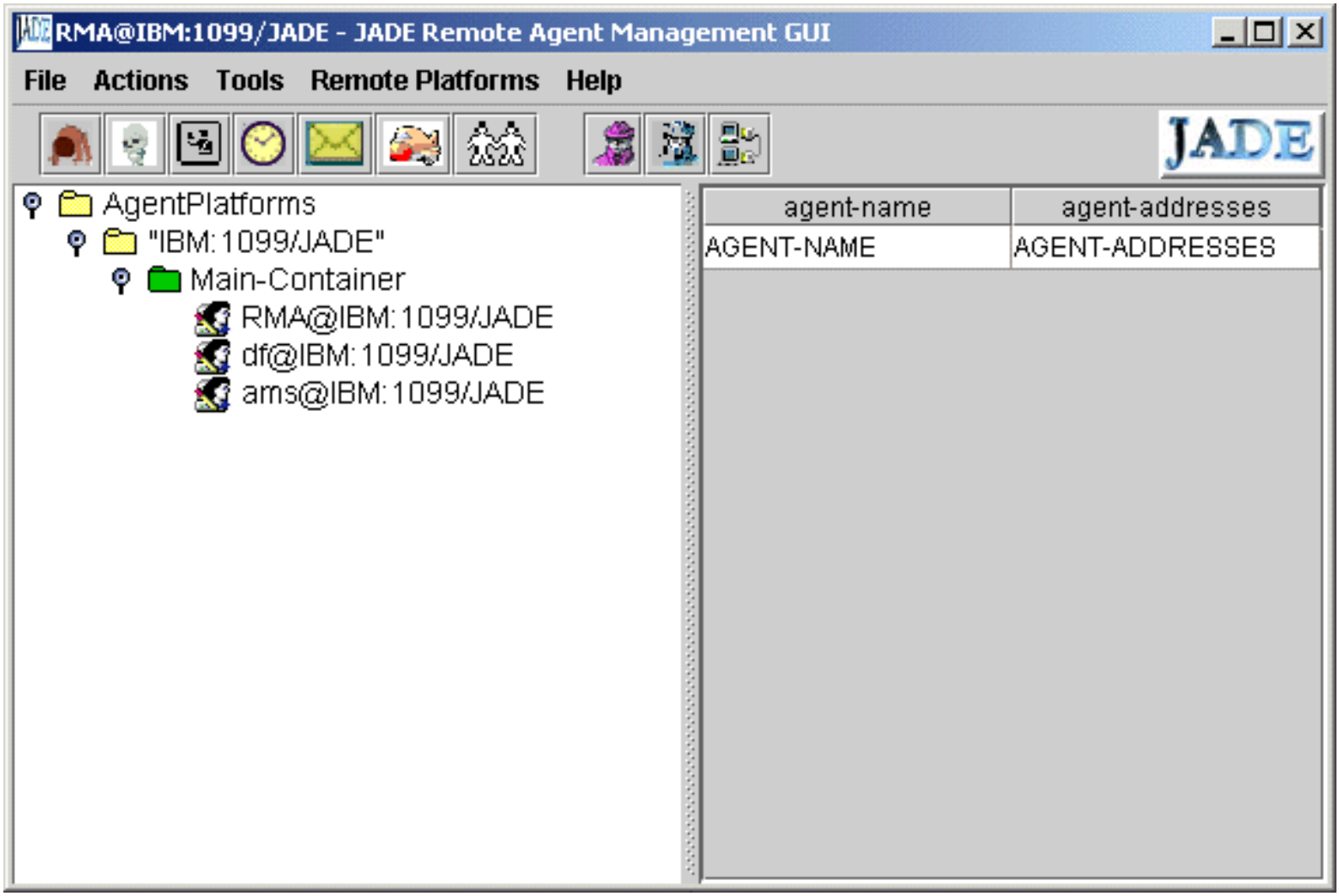
The JADE Agent Platform

The JADE platform is described in the JADE Programming Guide.

JADE Programming Guide

Platform Overview (excerpt from the guide)

When you boot the main JADE container with java jade.Boot -gui you get something like this.



This is the GUI of the RMA (Remote Management Agent). The RMA is the main tool for managing JADE.

Agents live in containers. Containers can be connected via RMI. They can be both local or remote. The Main container is associated with the RMI registry. The RMA can see all the containers. You can also have multiple RMA agents in different containers (but only one per container).

The JADE system itself is made up of agents. The two key ones are the AMS (Agent Management System agent) and the DF (directory facilitator). The characteristics of these agents are specified by FIPA. (see XC00023).

4 Agent Management Services

4.1 Directory Facilitator

4.1.1 Overview

A DF is a mandatory component of an AP that provides a ***yellow page***s directory service to agents. It is the trusted, benign custodian of the agent directory. It is trusted in the sense that it must strive to maintain an accurate, complete and timely list of agents. It is benign in the sense that it must provide the most current information about agents in its directory on a non-discriminatory basis to all authorised agents. At least one DF must be resident on each AP (the default DF). However, an AP may support any number of DFs and DFs may register with each other to form federations.

Every agent that wishes to publicise its services to other agents, should find an appropriate DF and request the registration of its agent description. There is no intended future commitment or obligation on the part of the registering agent implied in the act of registering. For example, an agent can refuse a request for a service which is advertised through a DF. Additionally, the DF cannot guarantee the validity or accuracy of the information that has been registered with it, neither can it control the life cycle of any agent. An object description must be supplied containing values for all of the mandatory parameters of the description. It may also supply optional and private parameters, containing non-FIPA standardised information that an agent developer might want included in the directory. The ***deregistration*** function has the consequence that there is no longer a commitment on behalf of the DF to broker information relating to that agent. At any time, and for any reason, the agent may request the DF to modify its agent description.

An agent may ***search in order to request information from a DF***. The DF does not guarantee the validity of the information provided in response to a search request, since the DF does not place any restrictions on the information that can be registered with it. However, the DF may restrict access to information in its directory and will verify all access permissions for agents which attempt to inform it of agent state changes.

The default DF on an AP has a reserved AID of:

(agent-identifier
:name df@hap
:addresses (sequence hap_transport_address))

4.1.2 Management Functions Supported by the Directory Facilitator

In order to access the directory of agent descriptions managed by the DF, each DF must be able to perform the following functions, when defined on the domain of objects of type df-agent-description in compliance with the semantics described in section6.1.2, Directory Facilitator Agent Description:

- register
- deregister
- modify
- search

4.1.3 Federated Directory Facilitators

The DF encompasses a search mechanism that searches first locally and then extends the search to other DFs, if allowed. The default search mechanism is assumed to be a depth-first search across DFs. For specific purposes, optional constraints can be used as described in section6.1.4, Search Constraints such as the number of answers (:df-search-results). The federation of DFs for extending searches can be achieved by DFs registering with each other with fipa-df as the value of the :type parameter in the service-description.

4.2 Agent Management System

4.2.1 Overview

An AMS is a mandatory component of the AP and only one AMS will exist in a single AP. ***The AMS is responsible for managing the operation of an AP, such as the creation of agents, the deletion of agents, deciding whether an agent can dynamically register with the AP and overseeing the migration of agents to and from the AP*** (if agent mobility is supported by the AP). Since different APs have different capabilities, the AMS can be queried to obtain a description of its AP. A life cycle is associated with each agent on the AP (see section5.1, Agent Life Cycle) which is maintained by the AMS.

The ***AMS represents the managing authority of an AP*** and if the AP spans multiple machines, then the AMS represents the authority across all machines. An AMS can request that an agent performs a specific management function, such as quit (that is, terminate all execution on its AP) and has the authority to forcibly enforce the function if such a request is ignored.

The AMS ***maintains an index of all the agents that are currently resident on an AP***, which includes the AID of agents. Residency of an agent on the AP implies that the agent has been registered with the AMS. Each agent, in order to comply with the FIPA reference model, must register with the AMS of its HAP. Registration with the AMS, implies authorisation to access the MTS of the AP in order to send or receive messages. The AMS will check the validity of the passed agent description and, in particular, the local uniqueness of the agent name in the AID.

Agent descriptions can be later modified at any time and for any reason. Modification is restricted by authorisation of the AMS. The life of an agent with an AP terminates with its deregistration from the AMS. After deregistration, the AID of that agent can be removed by the directory and can be made available to other agents who should request it.

Agent description can be searched with the AMS and access to the directory of ams-agent-descriptions is further controlled by the AMS; no default policy is specified by this specification.

The AMS is also the custodian of the AP description that can be retrieved by requesting the action get-description.

The AMS on an AP has a reserved AID of:

(agent-identifier

:name ams@hap

:addresses (sequence hap_transport_address))

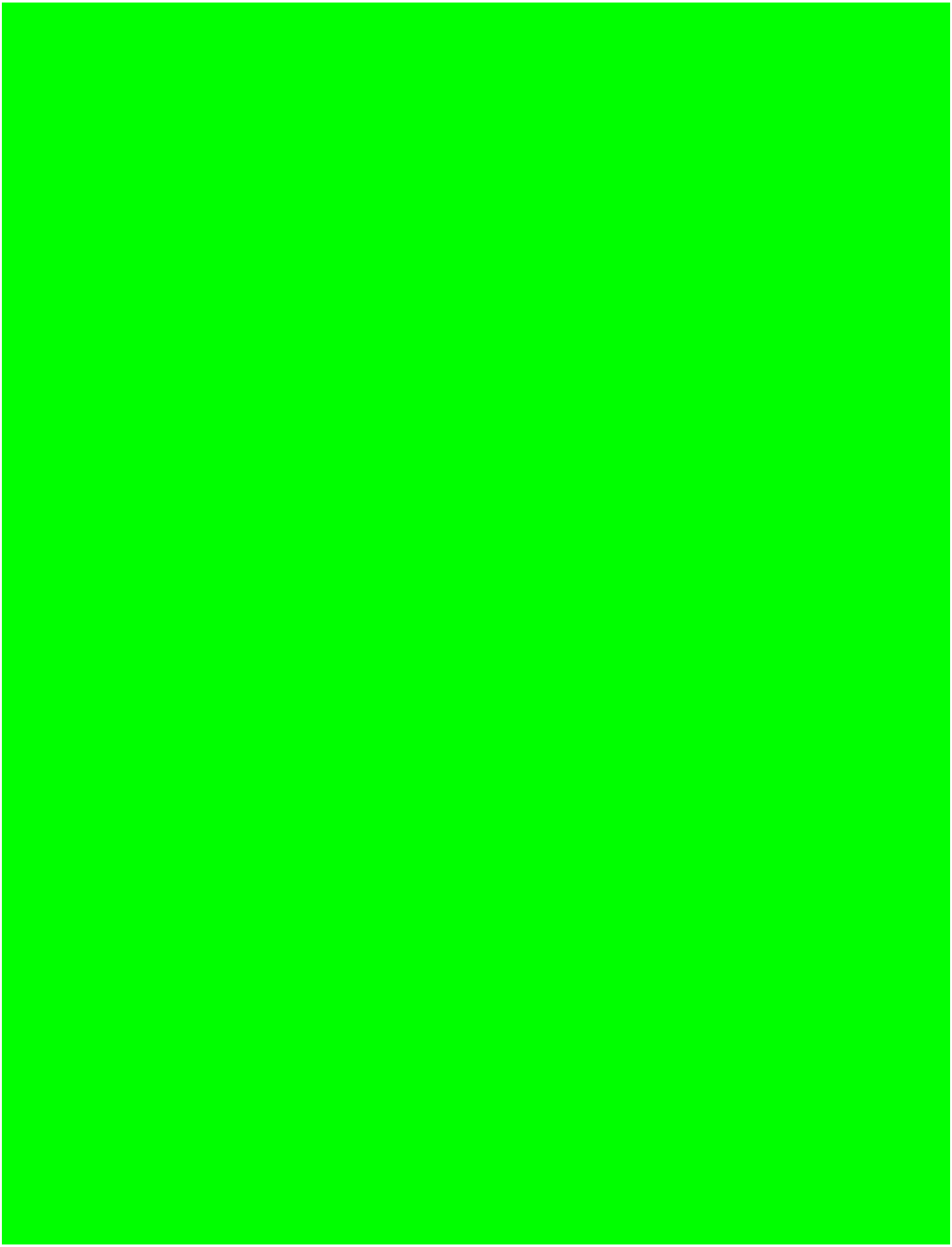
4.2.2 Management Functions Supported bSy the Agent Management System

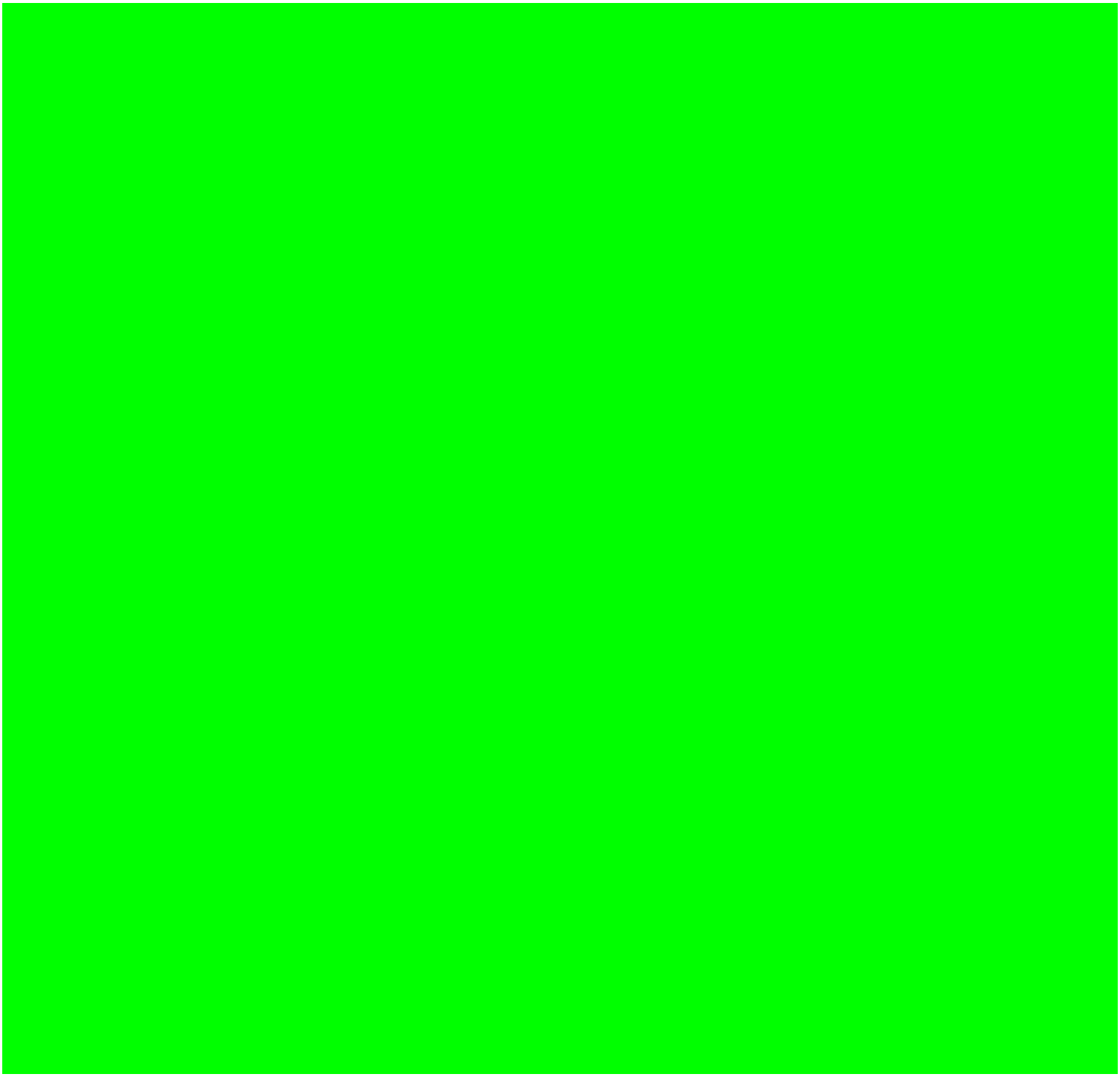
An AMS must be able to perform the following functions, in compliance with the semantics described in section6.1.5, Agent Management System Agent Description (the first four functions are defined within the scope of the AMS, only on the domain of objects of type ams-agent-description and the last on the domain of objects of type ap-description):

- register
- deregister
- modify
- search
- get-description

In addition to the management functions exchanged between the AMS and agents on the AP, the AMS can instruct the underlying AP to perform the following operations:

- Suspend agent,
- Terminate agent,
- Create agent,
- Resume agent execution,
- Invoke agent,
- Execute agent, and,
- Resource management.





JADE PROGRAMMER'S GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 4-September-2001. JADE 2.4

Authors: Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (*ex* CSELT *now* TILab)
Giovanni Rimassa (University of Parma)

JADE - Java Agent DEvelopment Framework is a framework to develop multi-agent systems in compliance with the FIPA specifications. JADE successfully passed the 1st FIPA interoperability test in Seoul (Jan. 99) and the 2nd FIPA interoperability test in London (Apr. 01).

TABLE OF CONTENTS

1 INTRODUCTION

This programmer's guide is complemented by the administrator's guide and the HTML documentation available in the directory jade/doc. If and where conflict arises between what is reported in the HTML documentation and this guide, preference should be given to the HTML documentation that is updated more frequently.

JADE (Java Agent Development Framework) is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has been fully coded in Java and an agent programmer, in order to exploit the framework, should code his/her agents in Java, following the implementation guidelines described in this programmer's guide.

This guide supposes the reader to be familiar with the FIPA standards[1], at least with the *Agent Management* specifications (FIPA no. 23), the *Agent Communication Language*, and the *ACL Message Structure* (FIPA no. 61).

JADE is written in Java language and is made of various Java packages, giving application programmers both ready-made pieces of functionality and abstract interfaces for custom, application dependent tasks. Java was the programming language of choice because of its many attractive features, particularly geared towards object-oriented programming in distributed heterogeneous environments; some of these features are Object Serialization, Reflection API and Remote Method Invocation (RMI).

JADE is composed of the following main packages.

`jade.core` implements the kernel of the system. It owns the `Agent` class that must be extended by application programmers; besides, a `Behaviour` class hierarchy is contained in `jade.core.behaviours` sub-package. Behaviours implement the tasks, or intentions, of an agent. They are logical activity units that can be composed in various ways to achieve complex execution patterns and that can be concurrently executed. Application programmers define agent operations writing behaviours and agent execution paths interconnecting them.

The `jade.lang` package has a sub-package for every language used in JADE. In particular, a `jade.lang.acl` sub-package is provided to process Agent Communication Language according to FIPA standard specifications. `jade.lang.sl` contains the SL-0 codec[2], both the parser and the encoder.

The `jade.onto` package contains a set of classes to support user-defined ontologies. It has a subpackage `jade.onto.basic` containing a set of basic concepts (i.e. Action, TruePredicate, FalsePredicate, …) that are usually part of every ontology, and a `BasicOntology` that can be joined with user-defined ontologies.

The `jade.domain` package contains all those Java classes that represent the Agent Management entities defined by the FIPA standard, in particular the AMS and DF agents, that provide life-cycle, white and yellow page services. The subpackage `jade.domain.FIPAAgentManagement` contains the FIPA-Agent-Management Ontology and all the classes representing its concepts. The subpackage

[1] See http://www.fipa.org/

[2] refer to FIPA document no. 8 for the specifications of the SL content language.

`jade.domain.JADEAgentManagement` contains, instead, the JADE extensions for Agent-Management (e.g. for sniffing messages, controlling the life-cycle of agents, …), including the Ontology and all the classes representing its concepts. The subpackage jade.domain.introspection contains the concepts used for the domain of discourse between the JADE tools (e.g. the Sniffer and the Introspector) and the JADE kernel.

The `jade.gui` package contains a set of generic classes useful to create GUIs to display and edit Agent-Identifiers, Agent Descriptions, ACLMessages, …

The `jade.mtp` package contains a Java interface that every Message Transport Protocol should implement in order to be readily integrated with the JADE framework, and the implementation of a set of these protocols.

`jade.proto` is the package that contains classes to model standard interaction protocols (i.e. *fipa-request*, *fipa-query*, *fipa-contract-net* and soon others defined by FIPA), as well as classes to help application programmers to create protocols of their own.

The `fipa` package contains the IDL module defined by FIPA for IIOP-based message transport.

Finally, the *jade.wrapper* package provides wrappers of the JADE higher-level functionalities that allows the usage of JADE as a library, where external Java applications launch JADE agents and agent containers (see also section 3.8).

JADE comes bundled with some tools that simplify platform administration and application development. Each tool is contained in a separate sub-package of `jade.tools`. Currently, the following tools are available:

- *Remote Management Agent, RMA* for short, acting as a graphical console for platform management and control. A first instance of an RMA can be started with a command line option ("*-gui*") , but then more than one GUI can be activated. JADE maintains coherence among multiple RMAs by simply multicasting events to all of them. Moreover, the RMA console is able to start other JADE tools.
- The *Dummy Agent* is a monitoring and debugging tool, made of a graphical user interface and an underlying JADE agent. Using the GUI it is possible to compose ACL messages and send them to other agents; it is also possible to display the list of all the ACL messages sent or received, completed with timestamp information in order to allow agent conversation recording and rehearsal.
- The *Sniffer* is an agent that can intercept ACL messages while they are in flight, and displays them graphically using a notation similar to UML sequence diagrams. It is useful for debugging your agent societies by observing how they exchange ACL messages.
- The *IntrospectorAgent* is a very useful tool that allows to monitor the life cycle of an agent and its exchanged ACL messages.
- The *SocketProxyAgent* is a simple agent, acting as a bidirectional gateway between a JADE platform and an ordinary TCP/IP connection. ACL messages, travelling over JADE proprietary transport service, are converted to simple ASCII strings and sent over a socket connection. Viceversa, ACL messages can be tunnelled via this TCP/IP connection into the JADE platform. This agent is useful, e.g. to handle network firewalls or to provide platform interactions with Java applets within a web browser.
- The *DF GUI* is a complete graphical user interface that is used by the default Directory Facilitator (DF) of JADE and that can also be used by every other DF that the user might need. In such a way, the user might create a complex network of domains and sub-domains of yellow pages. This GUI allows in a simple and intuitive

way to control the knowledge base of a DF, to federate a DF with other DF's, and to remotely control (register/deregister/modify/search) the knowledge base of the parent DF's and also the children DF's (implementing the network of domains and sub-domains).

JADE™ is a trade mark registered by CSELT[3].

2 JADE FEATURES

The following is the list of features that JADE offers to the agent programmer:

- Distributed agent platform. The agent platform can be split among several hosts (provided they can be connected via RMI). Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as Java threads and live within *Agent Containers* that provide the runtime support to the agent execution.
- Graphical user interface to manage several agents and agent containers from a remote host.
- Debugging tools to help in developing multi agents applications based on JADE.
- Intra-platform agent mobility, including state and code of the agent.
- Support to the execution of multiple, parallel and concurrent agent activities via the behaviour model. JADE schedules the agent behaviours in a non-preemptive fashion.
- FIPA-compliant Agent Platform, which includes the *AMS* (*Agent Management System*), the *DF* (*Directory Facilitator*), and the *ACC* (*Agent Communication Channel*). All these three components are automatically activated at the agent platform start-up.
- Many FIPA-compliant DFs can be started at run time in order to implement multi-domain applications, where a domain is a logical set of agents, whose services are advertised through a common facilitator. Each DF inherits a GUI and all the standard capabilities defined by FIPA (i.e. capability of registering, deregistering, modifying and searching for agent descriptions; and capability of federating within a network of DF's).
- Efficient transport of ACL messages inside the same agent platform. Infact, messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures. When crossing platform boundaries, the message is automatically converted to/from the FIPA compliant syntax, encoding, and transport protocol. This conversion is transparent to the agent implementers that only need to deal with Java objects.
- Library of FIPA interaction protocols ready to be used.
- Automatic registration and deregistration of agents with the AMS.
- FIPA-compliant naming service: at start-up agents obtain their GUID (Globally Unique Identifier) from the platform.
- Support for application-defined content languages and ontologies.
- InProcess Interface to allow external applications to launch autonomous agents.

3 CREATING MULTI-AGENT SYSTEMS WITH JADE

This chapter describes the JADE classes that support the development of multi-agent systems. JADE warrants syntactical compliance and, where possible, semantic compliance with FIPA specifications.

[3] Since March 2001, the name of the company is changed into TILab.

3.1 The Agent Platform

The standard model of an agent platform, as defined by FIPA, is represented in the following figure.

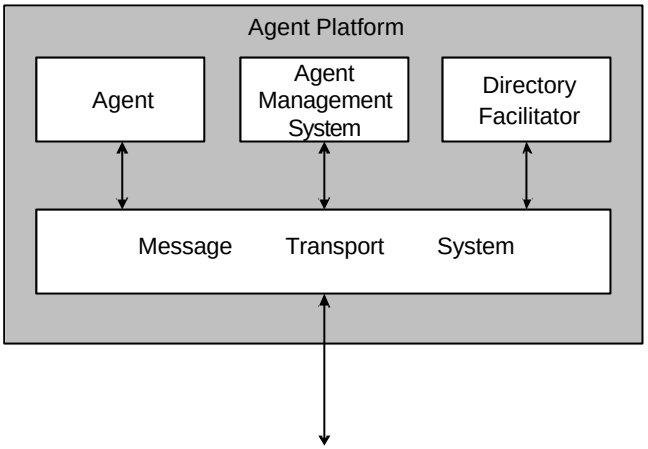


Figure 1 - Reference architecture of a FIPA Agent Platform

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

JADE fully complies with this reference architecture and when a JADE platform is launched, the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

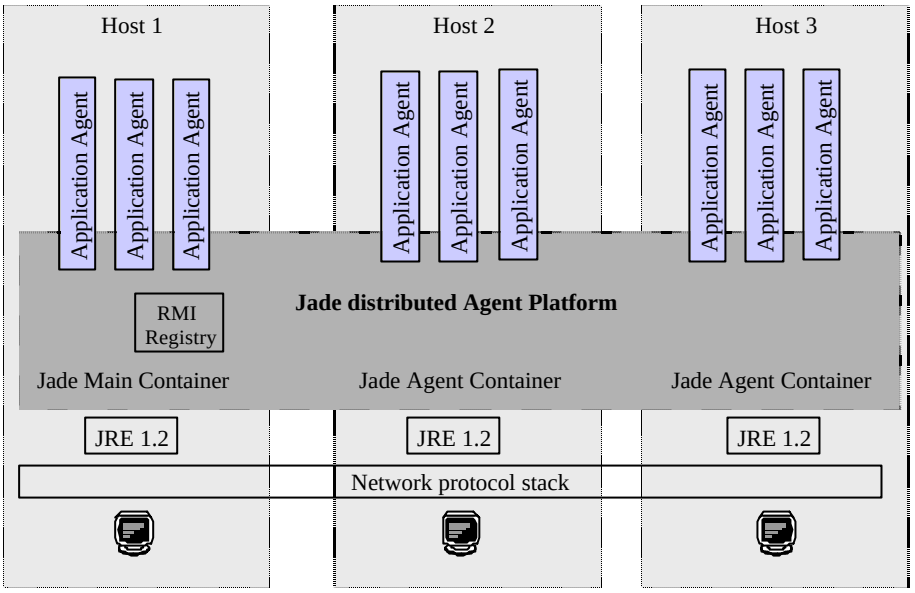


Figure 2 - JADE Agent Platform distributed over several containers

According to the FIPA specifications, DF and AMS agents communicate by using the `FIPA-SL0` content language, the `fipa-agent-management` ontology, and the `fipa-request` interaction protocol. JADE provides compliant implementations for all these components:

- the SL-0 content language is implemented by the class `jade.lang.sl.SL0Codec`. Automatic capability of using this language can be added to any agent by using the method `Agent.registerLanguage(SL0Codec.NAME, new SL0Codec());`
- concepts of the ontology (apart from Agent Identifier, implemented by `jade.core.AID`) are implemented by classes in the `jade.domain.FIPAAgentManagement` package. The `FIPAAgentManagementOntology` class defines the vocabulary with all the constant symbols of the ontology. Automatic capability of using this ontology can be added to any agent by using the fillowing code:
  `Agent.registerOntology(FIPAAgentManagementOntology.NAME, FIPAAgentManagementOntology.instance());`
- finally, the `fipa-request` interaction protocol is implemented as ready-to-use behaviours in the package `jade.proto`.

3.1.1 FIPA-Agent-Management ontology

Every class implementing a concept of the `fipa-agent-management` ontology is a simple collection of attributes, with public methods to read and write them, according to the frame based model that represents FIPA `fipa-agent-management` ontology concepts. The following convention has been used. For each attribute of the class, named `attrName` and of type `attrType`, two cases are possible:

1) The attribute type is a single value; then it can be read with `attrType getAttrName()` and written with `void setAttrName(attrType a)`, where every call to `setAttrName()` overwrites any previous value of the attribute.
2) The attribute type is a set or a sequence of values; then there is an `void addAttrName(attrType a)` method to insert a new value and a `void clearAllAttrName()` method to remove all the values (the list becomes empty). Reading is performed by a `Iterator getAllAttrName()` method that returns an `Iterator` object that allows the programmer to walk through the `List` and cast its elements to the appropriate type.

Refer to the HTML documentation for a complete list of these classes and their interface.

3.1.1.1 Basic concepts of the ontology

The package `jade.onto.basic` includes a set of classes that are commonly part of every ontology, such as `Action`, `TruePredicate`, `FalsePredicate`, `ResultPredicate`, … The `BasicOntology` can be joined to any user-defined ontology as described in section 3.6.

Notice that the `Action` class should be used to represent actions. It has a couple of methods to set/get the AID of the actor (i.e. the agent who should perform the action) and the action itself (e.g. *Register*/*Deregister*/*Modify*).

3.1.2 Simplified API to access DF and AMS services

JADE features described so far allow complete interactions between FIPA system agents and user defined agents, simply by sending and receiving messages as defined by the standard.

However, because those interactions have been fully standardized and because they are very common, the following classes allow to successfully accomplish this task with a simplified interface.

Two methods are implemented by the class `Agent` to get the AID of the default DF and AMS of the platform: `getDefaultDF()` and `getAMS()`.

3.1.2.1 DFServiceCommunicator

`jade.domain.DFServiceCommunicator` implements a set of static methods to communicate with a standard FIPA DF service (i.e. a yellow pages agent).

It includes methods to request `register`, `deregister`, `modify` and `search` actions from a DF. Each of this method has a version with all the needed parameters, and one with a subset of them where the omitted parameters are given default values.

Notice that these methods block every agent activity until the action is successfully executed or a `jade.domain.FIPAException` exception is thrown (e.g. because a `failure` message has been received by the DF), that is, until the end of the conversation.

In some cases, instead, it is more convenient to execute this task in a non-blocking way. The method `getNonBlockingBehaviour()` returns a non-blocking behaviour (of type `RequestFIPAServiceBehaviour`) that can be added to the agent behaviours, as usual, by using `Agent.addBehaviour()`. Several ways are available to get the result of this behaviour and the programmer can select one according to his preferred programming style:

- call `getLastMsg()` and `getSearchResults()` (both methods throw a `NotYetReadyException` if the task has not yet finished);
- create a `SequentialBehaviour` composed of two sub-behaviours: the first one is the returned `RequestFIPAServiceBehaviour`, while the second one is application-dependent and is executed only when the first is terminated;
- use the class `RequestFIPAServiceBehaviour` by extending it and overriding all the `handleXXX()` methods that handle the states of the `fipa-request` interaction protocol.

3.1.2.2 AMSServiceCommunicator

This class is dual of `DFServiceCommunicator` class, accessing services provided by a standard FIPA AMS agent and its interface completely corresponds the the `DFServiceCommunicator` one.

Notice that JADE calls automatically the `register` and `deregister` methods with the default AMS respectively before calling `setup()` method and just after `takeDown()` method returns; so there is no need for a normal programmer to call them.

However, under certain circumstances, a programmer might need to call its methods. To give some examples: when an agent wishes to register with the AMS of a remote agent platform, or when an agent wishes to modify its description by adding a private address to the set of its addresses, …

3.2 The Agent class

The `Agent` class represents a common base class for user defined agents. Therefore, from the programmer's point of view, a JADE agent is simply an instance of a user defined Java class that extends the base `Agent` class. This implies the inheritance of features to accomplish basic interactions with the agent platform (registration, configuration, remote management, …) and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, …).

The computational model of an agent is multitask, where tasks (or behaviours) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviours (refer to section 3.4 for implementation of behaviours). A scheduler, internal to the base `Agent` class and hidden to the programmer, automatically manages the scheduling of behaviours.

3.2.1 Agent life cycle

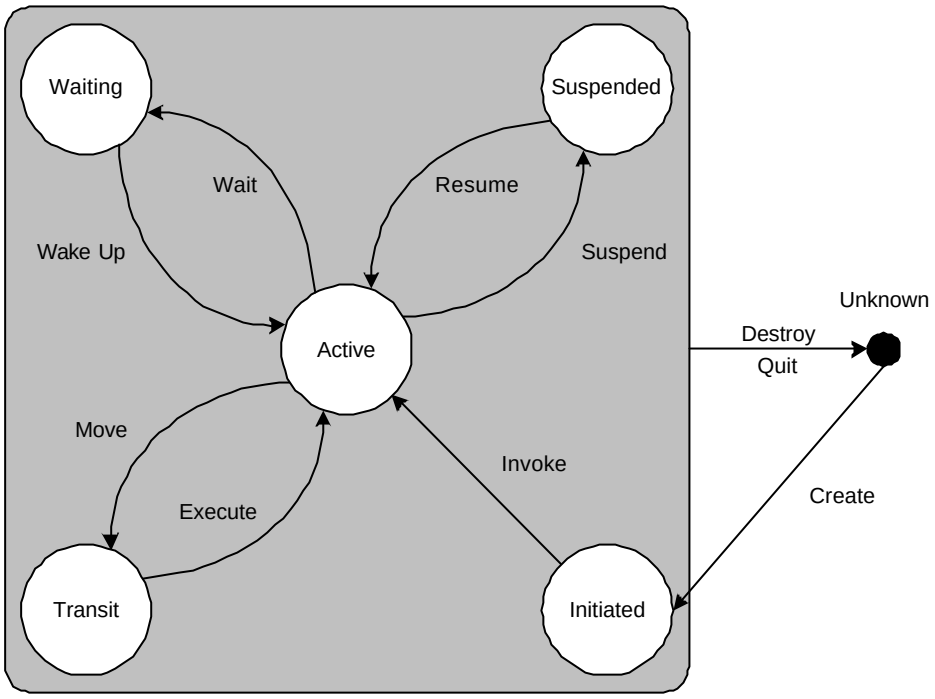


Figure 3 - Agent life-cycle as defined by FIPA.

A JADE agent can be in one of several states, according to Agent Platform Life Cycle in FIPA specification; these are represented by some constants in `Agent` class. The states are:

- **AP_INITIATED :** the Agent object is built, but hasn't registered itself yet with the AMS, has neither a name nor an address and cannot communicate with other agents.
- **AP_ACTIVE :** the Agent object is registered with the AMS, has a regular name and address and can access all the various JADE features.
- **AP_SUSPENDED :** the Agent object is currently stopped. Its internal thread is suspended and no agent behaviour is being executed.
- **AP_WAITING :** the Agent object is blocked, waiting for something. Its internal thread is sleeping on a Java monitor and will wake up when some condition is met (typically when a message arrives).
- **AP_DELETED :** the Agent is definitely dead. The internal thread has terminated its execution and the Agent is no more registered with the AMS.
- **AP_TRANSIT:** a mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.
- **AP_COPY:** this state is internally used by JADE for agent being cloned.
- **AP_GONE:** this state is internally used by JADE when a mobile agent has migrated to a new location and has a stable state.

The `Agent` class provides public methods to perform transitions between the various states; these methods take their names from a suitable transition in the Finite State Machine shown in FIPA specification *Agent Management*. For example, `doWait()` method puts the agent into AP_WAITING state from AP_ACTIVE state, `doSuspend()` method puts the agent into

AP_SUSPENDED state from AP_ACTIVE or AP_WAITING state, … Refer to the HTML documentation of the `Agent` class for a complete list of these `doXXX()` methods.

Notice that an agent is allowed to execute its behaviours (i.e. its tasks) only when it is in the AP_ACTIVE state. Take care that **if any behaviours call the `doWait()` method, then the whole agent and all its activities are blocked and not just the calling behaviour**. Instead, the `block()` method is part of the `Behaviour` class in order to allow suspending a single agent behaviour (see section 3.4 for details on the usage of behaviours).

3.2.1.1 Starting the agent execution

The JADE framework controls the birth of a new agent according to the following steps: the agent constructor is executed, the agent is given an identifier (see the HTML documentation for the `jade.core.AID` class), it is registered with the AMS, it is put in the AP_ACTIVE state, and finally the `setup()` method is executed. According to the FIPA specifications, an agent identifier has the following attributes:

- *a globally unique name*. By default JADE composes this name as the concatenation of the local name – i.e. the agent name provided on the command line – plus the '@' symbol, plus the home agent platform identifier – i.e. <hostname> ':' <port number of the JADE RMI registry> '/' 'JADE'); Tough in the case that the name of the platform is specified on the command line the agent name is constructed as a concatenation of the local name plus the '@' symbol plus the specified platform name.
- a set of agent addresses. Each agent inherits the transport addresses of its home agent platform;
- a set of resolvers, i.e. white page services with which the agent is registered.

The `setup()` method is therefore the point where any application-defined agent activity starts. The programmer has to implement the `setup()`method in order to initialise the agent. When the `setup()` method is executed, the agent has been already registered with the AMS and its Agent Platform state is *AP_ACTIVE*. The programmer should use this initialisation procedure to:

- (optional) if necessary, modify the data registered with the AMS (see section 3.1.2);
- (optional) set the description of the agent and its provided services and, if necessary, register the agent with one or more domains, i.e. DFs (see section 3.1.2);
- (necessary) add tasks to the queue of ready tasks using the method `addBehaviour()`. These behaviours are scheduled as soon as the `setup()` method ends;

The `setup()` method should add at least one behaviour to the agent. At the end of the `setup()` method, JADE automatically executes the first behaviour in the queue of ready tasks and then switch to the other behaviours in the queue by using a round-robin non-preemptive scheduler. The `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)` methods of the `Agent` class can be used to manage the task queue.

3.2.1.2 Stopping agent execution

Any behaviour can call the `Agent.doDelete()` method in order to stop agent execution.

The `Agent.takeDown()` method is executed when the agent is about to go to AP_DELETED state, i.e. it is going to be destroyed. The `takeDown()` method can be overridden by the programmers in order to implement any necessary cleanup. When this method is executed the agent is still registered with the AMS and can therefore send messages to other

agents, but just after the `takeDown()` method is completed, the agent will be de-registered and its thread destroyed. The intended purpose of this method is to perform application specific cleanup operations, such as de-registering with DF agents.

3.2.2 Inter-agent communication.

The `Agent` class also provides a set of methods for inter-agent communication. According to the FIPA specification, agents communicate via asynchronous message passing, where objects of the `ACLMessage` class are the exchanged payloads. See also section 3.3 for a description of the `ACLMessage` class. Some of the interaction protocols defined by FIPA are also available as ready-to-use behaviours that can be scheduled for agent activities; they are part of the `jade.proto` package.

The `Agent.send()` method allows to send an `ACLMessage`. The value of the `receiver` slot holds the list of the receiving agent IDs. The method call is completely transparent to where the agent resides, i.e. be it local or remote, it is the platform that takes care of selecting the most appropriate address and transport mechanism.

3.2.2.1 Accessing the private queue of messages.

All the messages received by an agent are put in its private queue by the agent platform. Several access modes have been implemented in order to get messages from this private queue:

- The message queue can be accessed in a blocking (using `blockingReceive()` method) or non-blocking way (using `receive()` method). The blocking version must be used very carefully because it ***causes the suspension of all the agent activities and in particular of all its Behaviours.*** The non-blocking version returns immediately `null` when the requested message is not present in the queue;
- both methods can be augmented with a pattern-matching capability where a parameter is passed that describes the pattern of the requested `ACLMessage`. Section 3.3.4 describes the `MessageTemplate` class;
- the blocking access can have a timeout parameter. It is a *long* that describes the maximum number of milliseconds that the agent activity should remain blocked waiting for the requested message. If the timeout elapses before the message arrives, the method returns `null`;
- the two behaviours `ReceiverBehaviour` and `SenderBehaviour` can be used to schedule agent tasks that requires receiving or sending messages.

3.2.3 Agents with a graphical user interface (GUI).

An application, that is structured as a Multi Agent System, still needs to interact with its users. So, it is often necessary to provide a GUI for at least some agents in the application. This need raises some problems, though, stemming from the mismatch between the autonomous nature of agents and the reactive nature of ordinary graphical user interfaces. When JADE is used, the *thread-per-agent* concurrency model of JADE agents must work together with the Swing concurrency model.

3.2.3.1 Java GUI concurrency model

In a Java Virtual Machine there is a single thread, called *Event Dispatcher Thread*, whose task is to continuously pick event objects (i.e. instances of `java.awt.AWTEvent` class) from the *System Event Queue* (which is an instance of `java.awt.EventQueue` class). Then the event dispatcher thread, among other things, calls the various listeners registered with the event source. The important observation is that all event listeners are executed within a single thread of control (the event dispatcher); from this follows the well known rule that the execution time of an event listener should be short (less than 0.1 s) to ensure interface responsiveness.
A very important Swing feature is the *Model/View* system to manage GUI updates. When a Swing control has some state (a `JCheckBox` has a `checked` flag, a `JList` holds elements, etc.), this state is kept in a *Model* object (of class `DefaultButtonModel`, `ListModel`, etc.). The model object provides commands to modify the state (e.g. to check or uncheck the checkbox, to add and remove elements from the list, etc.) and the Swing built-in notification mechanism updates the visual appearance of the GUI to reflect the state change. So, a `JCheckBox` object can change its look in two cases:

- An event from the user is received (e.g. a `MouseClick` event).
- Some other part of the program modifies the model object associated with the `JCheckBox`.

As stated in the *Java Tutorial* (*JFC/Swing* trail, *Threads and Swing* section), the Swing framework is not thread-safe, so any code that updates the GUI elements must be executed within the event dispatcher thread; since modifying a model object triggers an update of the GUI, it follows from the above that model objects also have to be manipulated just by the event dispatcher thread. The Swing framework provides a simple but general way to pass some user defined code to the Event Dispatcher thread: the `SwingUtilities` class exposes two static methods that accept a `Runnable` object, wrap it with a `RunnableEvent` and push it into the System Event Queue. The `invokeLater()` method puts the `Runnable` into the System Event Queue and returns immediately (behaving like an asynchronous inter-thread call), whereas the `invokeAndWait()` method puts the `Runnable` into the System Event Queue and blocks until the Event Dispatcher thread has processed the `RunnableEvent` (behaving like a synchronous inter-thread call). Moreover, the `invokeAndWait()` method can catch exceptions thrown within the `Runnable` object.

3.2.3.2 Performing an ACL message exchange in response to a GUI event.

When an agent is given a GUI, it often happens that the agent is requested to send a message because of a user action (e.g., the user clicks a pushbutton). The `ActionListener` of the button will be run within the Event Dispatcher thread, but the `Agent.send()` method should be called within the agent thread.

So:

In the event listener, add a new behaviour to the agent, which performs the necessary communication.

If the communication to perform is simply a message send operation, the `SenderBehaviour` class can be used, and the event handler will contain a line such as:

```
myAgent.addBehaviour(new SenderBehaviour(msgToSend));
```

If the communication operation is a message receive, the `ReceiverBehaviour` class can be used in the same way:

```
myAgent.addBehaviour(new ReceiverBehaviour(msgToRecv));
```

More generally, some complex conversation (e.g. a whole interaction conforming to an Interaction Protocol) could be started when the user acts on the GUI. The solution, again, is to add a new behaviour to the agent; this behaviour will extend the predefined JADE behaviours for Interaction Protocols or will be a custom complex behaviour.
The following code is extracted from the JADE RMA management agent. When the user wants to create a new agent, he or she operates on the RMA GUI (through the menu bar, the tool bar or a popup menu) to cause the execution of a `StartNewAgentAction` object, which calls the `newAgent()` method of the `rma` class. This method is implemented as follows:

```
      public void newAgent(String agentName, String className, Object
arg[], String containerName) {
        // Create a suitable content object for the ACL message ...
        // Set the :ontology slot of the message
    requestMsg.setOntology(JADEAgentManagementOntology.NAME);
        // Fill the message content with a List l, containing the
content object
    fillContent(requestMsg, l);
        addBehaviour(new AMSClientBehaviour("CreateAgent", requestMsg));
      }
```

The `AMSClientBehaviour` class is a private inner class of the `rma` class, that extends the `FipaRequestInitiatorBehaviour` and plays the *fipa-request* Interaction Protocol with the AMS agent. In this case, the `addBehaviour()` call and the specific class of the behaviour to add are completely encapsulated into the `rma` class. Various classes of the RMA GUI (mainly the action classes) hold a reference to the RMA agent and use it to call methods such as `newAgent()`. Notice that methods such as `newAgent()`don't really belong to the agent, because they don't access the agent state in any way. So, they are designed for being called from the outside (a different execution thread): in the following, these methods will be called *external methods*.
In general, it is not a good thing that an external software component maintain a direct object reference to an agent, because this component could directly call any public method of the agent (not just the external ones), skipping the asynchronous message passing layer and turning an autonomous agent into a server object, slave to its caller.
A better approach would be to gather all the external methods into an interface, implemented by the agent class. Then, an object reference of that interface will be passed to the GUI so that only the external methods will be callable from event handlers. The following pseudo code illustrates this approach:

```
interface RMAExternal {
  void newAgent(String agentName, String className, Object arg[], String
containerName);
  void suspendAgent(AID name);
  void resumeAgent(AID name);
  void killAgent(AID name);
  void killContainer(String name);
  void shutDownPlatform();
}
class MainWindow extends JFrame {
  private RMAExternal myRMA;
      public MainWindow(RMAExternal anRMA) {
    myRMA = anRMA;
  }
```

```
      // ...
}
class rma extends Agent implements RMAExternal {
  private MainWindow myGUI;
  protected void setup() {
    myGUI = new MainWindow(this);//Parameter 'this' typed as RMAExternal
    // ...
  }
}
```

With the schema above, the GUI will be able to call only the external methods of the RMA agent.

3.2.3.3 Modifying the GUI when an ACL message is received.

An agent can receive information from other agents through ACL messages: the *inform* FIPA communicative act serves just this purpose. If the agent has a GUI, it may often be the case that it wants to communicate the new information to its user by modifying the visual appearance of the GUI. According to the Model/View pattern, the new information should be used to modify some model objects, and Swing will take automatically care of updating the GUI. The `Agent.receive()` operation that read the message was executed within the agent thread, but any modification to Swing model objects must be performed from the Event Dispatcher thread. So:

In the agent behaviour, encapsulate all access to GUI model objects into a `Runnable` object and use `SwingUtilities.invokeXXX()` to submit the `Runnable` to the Event Dispatcher thread.

For example, when a new agent is born on a JADE platform, the AMS sends *inform* messages to all the active RMA agents; each one of them has to update its `AgentTree`, adding a node representing the new agent. The `rma` class holds a behaviour of the (inner and private) `AMSListener` class that continously receives `inform` messages from the AMS and dispatches them to suitable internal event handlers (it is basically a simple distributed event system over ACL messages). The handler corresponding to the *agent-born* event has the following code:

```
public void handle(AMSEvent ev) {
  AgentBorn ab = (AgentBorn)ev;
  String container = ab.getContainer();
  AID agent = ab.getAgent();
  myGUI.addAgent(container, agent);
}
```

The `addAgent()` method of the class `MainWindow` is the following:

```
public void addAgent(final String containerName, final AID agentID) {
  // Add an agent to the specified container
  Runnable addIt = new Runnable() {
  public void run() {
    String agentName = agentID.getName();
    AgentTree.Node node = tree.treeAgent.createNewNode(agentName, 1);
    Iterator add = agentID.getAllAddresses();
    String agentAddresses = "";
    while(add.hasNext())
      agentAddresses = agentAddresses + add.next() + " ";
      tree.treeAgent.addAgentNode((AgentTree.AgentNode)node,
containerName, agentName, agentAddresses, "FIPAAGENT");
```

```
    }
  };
  SwingUtilities.invokeLater(addIt);
}
```

As can be seen from the above code, all the accesses to the agent tree are encapsulated inside a `Runnable` that is submitted for execution to the Event Dispatcher thread using the `SwingUtilities.invokeLater()` method. The whole process of `Runnable` creation and submission is contained within the `addAgent()` method of the `MainWindow` class, so that the rma agent does not directly deal with Swing calls (it does not even have to import Swing related classes).
If we consider the whole `MainWindow` as an active object whose thread is the Event Dispatcher thread, then the `addAgent()` method is clearly an external method and this approach mirrors exactly the technique used in the section above. However, since the GUI is not to be seen as an autonomous software component, the choice of using external methods or not is just a matter of software structure, without particular conceptual meaning.

3.2.3.4 Support for building GUI enabled agents in JADE.

Because it is quite common having agents with a GUI, JADE includes the class `jade.gui.GuiAgent` for this specific purpose. This class is a simple extension of the `jade.core.Agent` class: at the start-up (i.e. when the method `setup()` is executed) it instantiates an ad-hoc behaviour that manages a queue of `jade.gui.GuiEvent` event objects that can be received by other threads. This behaviour is of course hidden to the programmer who needs only to implement the application-specific code relative to each event. In detail, the following operations must be performed.

A thread (in particular the GUI) wishing to notify an event to an agent should create a new object of type `jade.gui.GuiEvent` and pass it as a parameter to the call of the method `postGuiEvent()` of the `jade.gui.GuiAgent` object. After the method `postGuiEvent()` is called, the agent reacts by waking up all its active behaviours, and in particular the behaviour above mentioned that causes the agent thread to execute the method `onGuiEvent()`. Notice that an object `GuiEvent` has two mandatory attributes (i.e. the source of the event and an integer identifying the type of event) and an optional list of parameters[4] that can be added to the event object.

As a consequence, an agent wishing to receive events from another thread (in particular its GUI) should define the types of events it intends to receive and then implement the method `onGuiEvent()`. In general, this method is a big switch, one case for each type of event. The example *mobile*, distributed with JADE, is a good example of this feature.

In order to explain further the previous concepts, in the following are reported some interesting points of the code of the example concerning the `MobileAgent`.

File MobileAgent.java

[4] The type of each parameter must extend *java.lang.Object;* therefore primitive objects (e.g. *int*) should before be wrapped into appropriate objects (e.g. *java.lang.Integer).*

```
public class MobileAgent extends GuiAgent {

  ……

  // These constants are used by the Gui to post Events to the
  Agent
  public static final int EXIT = 1000;
  public static final int MOVE_EVENT = 1001;
  public static final int STOP_EVENT = 1002;
  public static final int CONTINUE_EVENT = 1003;
  public static final int REFRESH_EVENT = 1004;
  public static final int CLONE_EVENT = 1005;

  ……
public void setup() {
    ……
    // creates and shows the GUI
    gui = new MobileAgentGui(this);
    gui.setVisible(true);
    ……
  }
  ……

  // AGENT OPERATIONS FOLLOWING GUI EVENTS
  protected void onGuiEvent(GuiEvent ev)
  {
       switch(ev.getType())
       {
       case EXIT:
            gui.dispose();
            gui = null;
            doDelete();
            break;
       case MOVE_EVENT:
             Iterator moveParameters = ev.getAllParameter();
             nextSite =(Location)moveParameters.next();
            doMove(nextSite);
            break;
       case CLONE_EVENT:
            Iterator cloneParameters = ev.getAllParameter();
            nextSite =(Location)cloneParameters.next();
```

```
                doClone(nextSite,"clone"+cnt+"of"+getName());
                break;
          case STOP_EVENT:
                stopCounter();
                break;
          case CONTINUE_EVENT:
                continueCounter();
                break;
          case REFRESH_EVENT:
                addBehaviour(new
                GetAvailableLocationsBehaviour(this));
                break;
          }

    }
}
```

File MobileAgentGui.java

```
package examples.mobile;

public class MobileAgentGui extends JFrame implements
ActionListener
{
  private MobileAgent  myAgent;
  ……

  // Constructor
  MobileAgentGui(MobileAgent a)
  {
        super();
        myAgent = a;

        ……

        JButton pauseButton = new JButton("STOP COUNTER");
        pauseButton.addActionListener(this);
        Jbutton continueButton = new JButton("CONTINUE
        COUNTER");
        continueButton.addActionListener(this);
        …
        JButton b = new JButton(REFRESHLABEL);
```

```
        b.addActionListener(this);
        …
        b = new JButton(MOVELABEL);
        b.addActionListener(this);
        …
        b = new JButton(CLONELABEL);
        b.addActionListener(this);
        …
        b = new JButton(EXITLABEL);
        b.addActionListener(this);
        ……
    }
      ……

    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
        // MOVE
        if (command.equalsIgnoreCase(MOVELABEL)) {
          Location dest;
          int sel = availableSiteList.getSelectedRow();
          if (sel >= 0)
            dest = availableSiteListModel.getElementAt(sel);
          else
            dest = availableSiteListModel.getElementAt(0);

        GuiEvent ev = new
        GuiEvent((Object)this,myAgent.MOVE_EVENT);
        ev.addParameter(dest);
         myAgent.postGuiEvent(ev);
        }
        // CLONE
        else if (command.equalsIgnoreCase(CLONELABEL)) {
          Location dest;
          int sel = availableSiteList.getSelectedRow();
          if (sel >= 0)
            dest = availableSiteListModel.getElementAt(sel);
          else
            dest = availableSiteListModel.getElementAt(0);
          GuiEvent ev = new
                GuiEvent((Object)this,myAgent.CLONE_EVENT);
```

```
            ev.addParameter(dest);
                 myAgent.postGuiEvent(ev);
          }
          // EXIT
          else if (command.equalsIgnoreCase(EXITLABEL)) {
                GuiEvent ev = new GuiEvent(null,myAgent.EXIT);
                myAgent.postGuiEvent(ev);
          }
          else if (command.equalsIgnoreCase(PAUSELABEL)) {
             GuiEvent ev = new GuiEvent(null,myAgent.STOP_EVENT);
             myAgent.postGuiEvent(ev);
          }
          else if (command.equalsIgnoreCase(CONTINUELABEL)){
       GuiEvent ev = new GuiEvent(null,myAgent.CONTINUE_EVENT);
        myAgent.postGuiEvent(ev);
       }
          else if (command.equalsIgnoreCase(REFRESHLABEL)) {
      GuiEvent ev = new GuiEvent(null,myAgent.REFRESH_EVENT);
      myAgent.postGuiEvent(ev);
          }
      }
      ……
  }
```

3.2.4 Agent with parameters and launching agents

A list of arguments can be passed to an `Agent` and they can be retrieved by calling the method `Object[] getArguments()`. Notice that the arguments are transient and they do not migrate with the agent neither they are cloned with the agent.

There are three ways of launching an agent:

- a list of agents can be specified on the command line, by using the syntax described in the Administrator's Guide. Arguments, embedded within parenthesis, can be passed to each agent. This is the most common option and the option that best matches the theoretical requirement of agent autonomy.

- an agent can be launched by an administrator by using the RMA (Remote Monitoring Agent) GUI, as described in the Administrator's Guide. Arguments, embedded within parenthesis, can be passed to each agent.

- finally, an agent can also be launched by any external Java program by using the InProcess Interface as described in section 3.8

3.3 Agent Communication Language (ACL) Messages

The class `ACLMessage` represents ACL messages that can be exchanged between agents. It contains a set of attributes as defined by the FIPA specifications.

An agent willing to send a message should create a new `ACLMessage` object, fill its attributes with appropriate values, and finally call the method `Agent.send()`. Likewise, an agent willing to receive a message should call `receive()` or `blockingReceive()` methods, both implemented by the `Agent` class and described in section 3.2.2.

Sending or receiving messages can also be scheduled as independent agent activities by adding the behaviours `ReceiverBehaviour` and `SenderBehaviour` to the agent queue of tasks.

All the attributes of the `ACLMessage` object can be accessed via the `set/get<Attribute>()` access methods. All attributes are named after the names of the parameters, as defined by the FIPA specifications. Those parameters whose type is a set of values (like receiver, for instance) can be accessed via the methods `add/getAll<Attribute>()` where the first method adds a value to the set, while the second method returns an `Iterator` over all the values in the set. Notice that all the `get()` methods return `null` when the attribute has not been yet set.

Furthermore, this class also defines a set of constants that should be used to refer to the FIPA performatives, i.e. `REQUEST`, `INFORM`, etc. When creating a new `ACLMessage` object, one of these constants must be passed to `ACLMessage` class constructor, in order to select the message performative. The `reset()` method resets the values of all message fields.

The `toString()` method returns a string representing the message. This method should be just used for debugging purposes.

3.3.1 Support to reply to a message

According to FIPA specifications, a reply message must be formed taking into account a set of well-formed rules, such as setting the appropriate value for the attribute *in-reply-to*, using the same *conversation-id*, etc. JADE helps the programmer in this task via the method `createReply()` of the `ACLMessage` class. This method returns a new `ACLMessage` object that is a valid reply to the current one. Then, the programmer only needs to set the application-specific communicative act and message content.

3.3.2 Support for Java serialisation and transmission of a sequence of bytes

Some applications may benefit from transmitting a sequence of bytes over the content of an `ACLMessage`. A typical usage is passing Java objects between two agents by exploiting the Java serialization. The `ACLMessage` class supports the programmer in this task by allowing the usage of *Base64* encoding through the two methods `setContentObject()` and `getContentObject()`. Refer to the HTML documentation of the JADE API and to the examples in `examples/Base64` directory for an example of usage of this feature.

It must be noticed that this feature does not comply to FIPA and that any agent platform can recognize automatically the usage of Base64 encoding[5], so the methods must appropriately used

[5] The implementation of this feature uses the source code contained within the src/starlight directory. This code is covered by the GNU General Public License, as decided by the copyright owner Kevin Kelley. The GPL license itself has been included as a text file named COPYING in the same directory. If the programmer does not need any support for Base64 encoding, then this code is not necessary and can be removed.

by the programmers and should suppose that communicating agents know a-priori the usage of these methods.

3.3.3 The ACL Codec

Under normal conditions, agents never need to call explicitly the codec of the ACL messages because it is done automatically by the platform. However, when needed for some special circumstances, the programmer should use the methods provided by the class `StringACLCodec` to parse and encode ACL messages in String format.

3.3.4 The MessageTemplate class

The JADE behaviour model allows an agent to execute several parallel tasks. However any agent should be provided with the capability of carrying on also many simultaneous conversations. Because the queue of incoming messages is shared by all the agent behaviours, an access mode to that queue based on pattern matching has been implemented (see 3.2.2.1).

The `MessageTemplate` class allows to build patterns to match ACL messages against. Using the methods of this class the programmer can create one pattern for each attribute of the `ACLMessage`. Elementary patterns can be combined with AND, OR and NOT operators, in order to build more complex matching rules. In such a way, the queue of incoming ACL messages can be accessed via pattern-matching rather than FIFO.

The user can also define application specific patterns extending the MatchExpression interface in order to provide a new `match()` method to use in the pattern matching phase.

The example WaitAgent in the MessageTemplate directory of the package examples, shows a way to create an application-specific MessageTemplate:

```
public class WaitAgent extends Agent {
  Class myMatchExpression implements
  MessageTemplate.MatchExpression {
    List senders;
    myMatchExpression(List l){
       senders = l;
    }
    public boolean match(ACLMessage msg){
      AID sender = msg.getSender();
      String name = sender.getName();
      Iterator it_temp = senders.iterator();
      boolean out = false;

      while(it_temp.hasNext() && !out){
       String tmp = ((AID)it_temp.next()).getName();
       if(tmp.equalsIgnoreCase(name))
             out = true;
      }
      return out;
```

```
            }

      }
      class WaitBehaviour extends SimpleBehaviour {
        public WaitBehaviour(Agent a, MessageTemplate mt) {
            ……}
        public void action() {
            ……
            ACLMessage  msg = blockingReceive(template);
            ……
        }
        ……
      } //End class WaitBehaviour

      protected void setup() {
             ……
            ArrayList sender = ……
            myMatchExpression me = new myMatchExpression(sender);
            MessageTemplate myTemplate = new MessageTemplate(me);

            MessageTemplate mt =
MessageTemplate.and(myTemplate,MessageTemplate.MatchPerformative(
ACLMessage.REQUEST));

            WaitBehaviour behaviour = new  WaitBehaviour(this,mt);
            addBehaviour(behaviour);
            }catch(java.io.IOException e){
                      e.printStackTrace();
      }
      }
   }//end class WaitAgent
```

3.4 The agent tasks. Implementing Agent behaviours

An agent must be able to carry out several concurrent tasks in response to different external events. In order to make agent management efficient, every JADE agent is composed of a single execution thread and all its tasks are modelled and can be implemented as `Behaviour` objects. Multi-threaded agents can also be implemented but no specific support (except synchronizing the ACL message queue) is provided by JADE.

The developer who wants to implement an agent-specific task should define one or more `Behaviour` subclasses, instantiate them and add the behaviour objects to the agent task list. The `Agent` class, which must be extended by agent programmers, exposes two methods: `addBehaviour(Behaviour)` and `removeBehaviour(Behaviour)`, which allow to

manage the ready tasks queue of a specific agent. Notice that behaviours and sub-behaviours can be added whenever is needed, and not only within `Agent.setup()` method. Adding a behaviour should be seen as a way to spawn a new (cooperative) execution thread within the agent.

A scheduler, implemented by the base `Agent` class and hidden to the programmer, carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue, executing a `Behaviour`-derived class until it will release control (this happens when `action()` method returns). If the task relinquishing the control has not yet completed, it will be rescheduled the next round. A behaviour can also block, waiting for a message to arrive. In detail, the agent scheduler executes `action()` method of each behaviour present in the ready behaviours queue; when `action()` returns, the method `done()` is called to check if the behaviour has completed its task. If so, the behaviour object is removed from the queue.

Behaviours work just like co-operative threads, but there is no stack to be saved. ***Therefore, the whole computation state must be maintained in instance variables of the Behaviour and its associated Agent***.

In order to avoid an active wait for messages (and, as a consequence, a waste of CPU time), every single `Behaviour` is allowed to block its computation. ***The method `block()` puts the behaviour in a queue of blocked behaviours as soon as the `action()` method returns.*** Notice, therefore, that the blocking effect is not achieved immediately after calling the `block()` method, but just after returning from the `action()` method. All blocked behaviours are rescheduled as soon as a new message arrives, therefore the programmer must take care of blocking again a behaviour if it was not interested in the arrived message. Moreover, a behaviour object can block itself for a limited amount of time passing a timeout value to `block()` method. In future releases of JADE, more wake up events will be probably considered.

Because of the non preemptive multitasking model chosen for agent behaviours, agent programmers must avoid to use endless loops and even to perform long operations within `action()` methods. Remember that when some behaviour's `action()` is running, no other behaviour can go on until the end of the method (of course this is true only with respect to behaviours of the same agent: behaviours of other agents run in different Java threads and can still proceed independently).

Besides, since no stack contest is saved, every time `action()` method is run from the beginning: there is no way to interrupt a behaviour in the middle of its `action()`, yield the CPU to other behaviours and then start the original behaviour back from where it left.

For example, suppose a particular operation `op()` is too long to be run in a single step and is therefore broken in three sub-operations, named `op1()`,`op2()` and `op3()`. To achieve desired functionality one must call `op1()` the first time the behaviour is run, `op2()` the second time and `op3()` the third time, after which the behaviour must be marked as terminated. The code will look like the following:

```
public class my3StepBehaviour {
  private int state = 1;
  private boolean finished = false;

  public void action() {
    switch (state) {
      case 1: { op1(); state++; break; }
```

```
        case 2: { op2(); state++; break; }
        case 3: { op3(); state=1; finished = true; break; }
       }
    }

    public boolean done() {
      return finished;
    }
  }
```

Following this idiom, agent behaviours can be described as finite state machines, keeping their whole state in their instance variables.

When dealing with complex agent behaviours (as agent interaction protocols) using explicit state variables can be cumbersome; so JADE also supports a compositional technique to build more complex behaviours out of simpler ones.

The framework provides ready to use `Behaviour` subclasses that can contain sub-behaviours and execute them according to some policy. For example, a `SequentialBehaviour` class is provided, that executes its sub-behaviours one after the other for each `action()` invocation.

The following figure is an annotated UML class diagram for JADE behaviours.

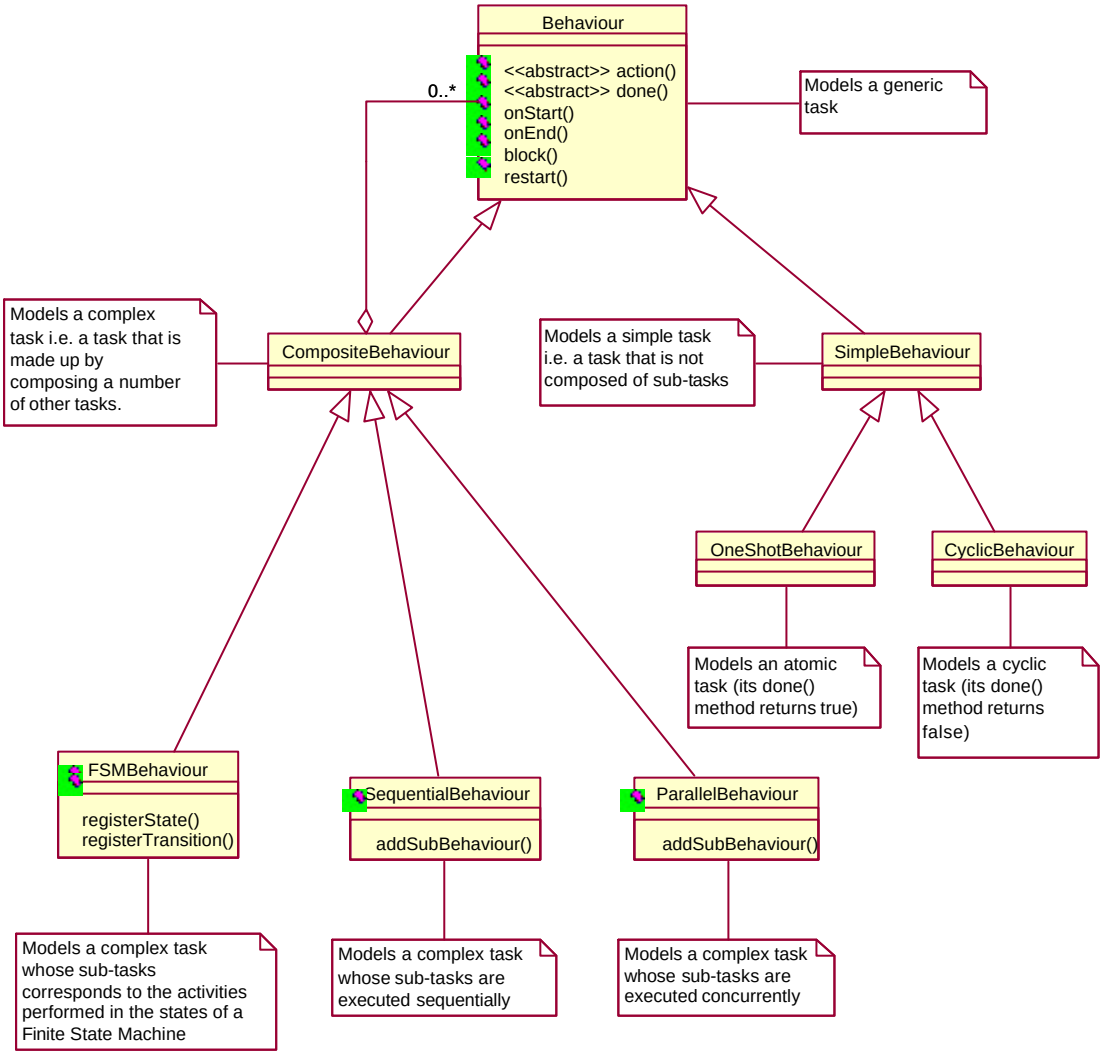


Figure 4 - UML Model of the Behaviour class hierarchy

Starting from the basic class `Behaviour`, a class hierarchy is defined in the `jade.core.behaviour` package of the JADE framework.
A complete description of all these classes follows.

3.4.1 class Behaviour

This abstract class provides an abstract base class for modelling agent tasks, and it sets the basis for behaviour scheduling as it allows for state transitions (i.e. starting, blocking and restarting a Java behaviour object).

The `block()` method allows to block a behaviour object until some event happens (typically, until a message arrives). This method leaves unaffected the other behaviours of an agent, thereby allowing finer grained control on agent multitasking. This method puts the behaviour in a queue of blocked behaviours and takes effect as soon as `action()` returns. All blocked behaviours are rescheduled as soon as a new message arrives. Moreover, a behaviour

object can block itself for a limited amount of time passing a timeout value to `block()` method, expressed in milliseconds. In future releases of JADE, more wake up events will be probably considered. A behaviour can be explicitly restarted by calling its `restart()` method.

Summarizing, a blocked behaviour can resume execution when one of the following three conditions occurs:

1. An ACL message is received by the agent this behaviour belongs to.
2. A timeout associated with this behaviour by a previous `block()` call expires.
3. The `restart()` method is explicitly called on this behaviour.

The Behaviour class also provides two placeholders methods, named `onStart()` and `onEnd()`. These methods can be overridden by user defined subclasses when some actions are to be executed before and after running behaviour execution.

onEnd() returns an int that represents a termination value for the behaviour.

It should be noted that onEnd() is called after the behaviour has completed and has been removed from the pool of agent behaviours. Therefore calling reset() inside onEnd() is not sufficient to cyclically repeat the task represented by that behaviour; besides that the behaviour should be added again to the agent as in the following example

```
public int onEnd() {
  reset();
  myAgent.addBehaviour(this);
  return 0;
}
```

This class provides also a couple of methods to get and set a *DataStore* for the behaviour. The *DataStore* can be a useful repository for exchanging data between behaviours, as done, for instance, by the classes *jade.proto.AchieveREInitiator/Responder*. **Notice that the DataStore is cleaned and all the contained data are lost when the behaviour is reset.**

3.4.2 class SimpleBehaviour

This abstract class models simple atomic behaviours. Its `reset()` method does nothing by default, but it can be overridden by user defined subclasses.

3.4.3 class OneShotBehaviour

This abstract class models atomic behaviours that must be executed only once and cannot be blocked. So, its `done()` method always returns `true`.

3.4.4 class CyclicBehaviour

This abstract class models atomic behaviours that must be executed forever. So its `done()` method always returns `false`.

3.4.5 class CompositeBehaviour

This abstract class models behaviours that are made up by composing a number of other behaviours (children). So the actual operations performed by executing this behaviour are not defined in the behaviour itself, but inside its children while the composite behaviour takes only

care of children scheduling according to a given policy[6].

In particular the CompositeBehaviour class only provides a common interface for children scheduling, but does not define any scheduling policy. This scheduling policy must be defined by subclasses (SequentialBehaviour, ParallelBehaviour and FSMBehaviour). A good programming practice is therefore to use only CompositeBehaviour sub-classes, unless some special children scheduling policy is needed (e.g. a `PriorityBasedCompositeBehaviour` should extend `CompositeBehaviour` directly).

Notice that this class was renamed since JADE 2.2 and it was previously called `ComplexBehaviour.`

3.4.6 class SequentialBehaviour

This class is a `CompositeBehaviour` that executes its sub-behaviours sequentially and terminates when all sub-behaviours are done. Use this class when a complex task can be expressed as a sequence of atomic steps (e.g. do some computation, then receive a message, then do some other computation).

3.4.7 class ParallelBehaviour

This class is a `CompositeBehaviour` that executes its sub-behaviours concurrently and terminates when a particular condition on its sub-behaviours is met. Proper constants to be indicated in the constructor of this class are provided to create a `ParallelBehaviour` that ends when all its sub-behaviours are done, when any one among its sub-behaviour terminates or when a user defined number *N* of its sub-behaviours have finished. Use this class when a complex task can be expressed as a collection of parallel alternative operations, with some kind of termination condition on the spawned subtasks.

Notice that this class was renamed since JADE 2.2 and it was previously called `NonDeterministicBehaviour.`

3.4.8 class FSMBehaviour

This class is a `CompositeBehaviour` that executes its children according to a Finite State Machine defined by the user. More in details each child represents the activity to be performed within a state of the FSM and the user can define the transitions between the states of the FSM. When the child corresponding to state S_i completes, its termination value (as returned by the onEnd() method) is used to select the transition to fire and a new state S_j is reached. At next round the child corresponding to S_j will be executed. Some of the children of an FSMBehaviour can be registered as final states. The FSMBehaviour terminates after the completion of one of these children.

Refer to the javadoc documentation of the JADE APIs for a detailed description on how to describe a Finite State Machine both at execution-time or static compilation time.

[6] Each time the `action`() method of a complex behaviour is called this results in calling the `action`() method of one of its children. The scheduling policy determines which children to select at each round.

3.4.9 class SenderBehaviour

Encapsulates an atomic unit which realises the "send" action. It extends `OneShotBehaviour` class and so it is executed only once. An object with this class must be given the ACL message to send at construction time.

3.4.10 class ReceiverBehaviour

Encapsulates an atomic operation which realises the "receive" action. Its action terminates when a message is received. If the message queue is empty or there is no message matching the `MessageTemplate` parameter, `action()` method calls `block()` and returns. The received message is copied into a user specified `ACLMessage`, passed in the constructor. Two more constructors take a timeout value as argument, expressed in milliseconds; a `ReceiverBehaviour` created using one of these two constructors will terminate after the timeout has expired, whether a suitable message has been received or not. An `Handle` object is used to access the received ACL message; when trying to retrieve the message suitable exceptions can be thrown if no message is available or the timeout expired without any useful reception.

3.4.11 class WakerBehaviour

This abstract class implements a OneShot task that must be executed only once just after a given timeout is elapsed.

3.4.12 Examples

In order to explain further the previous concepts, an example is reported in the following. It illustrates the implementation of two agents that, respectively, send and receive messages. The behaviour of the `AgentSender` extend the `SimpleBehaviour` class so it simply sends some messages to the receiver and than kills itself. The `AgentReceiver` has instead a behaviour that extends `CyclicBehaviour` class and shows different kinds to receive messages.

File AgentSender.java

```
package examples.receivers;

import java.io.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class AgentSender extends Agent {

  protected void setup() {
    addBehaviour(new SimpleBehaviour(this) {
      private boolean finished = false;
      public void action() {
        try{
```

```
            System.out.println("\nEnter responder agent name: ");
            BufferedReader buff = new BufferedReader(new
                    InputStreamReader(System.in));
            String responder = buff.readLine();
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(new AID(responder));
            msg.setContent("FirstInform");
            send(msg);
            System.out.println("\nFirst INFORM sent");
            doWait(5000);
            msg.setLanguage("PlainText");
            msg.setContent("SecondInform");
            send(msg);
            System.out.println("\nSecond INFORM sent");
            doWait(5000);
           // same that second
           msg.setContent("\nThirdInform");
           send(msg);
           System.out.println("\nThird INFORM sent");
           doWait(1000);
           msg.setOntology("ReceiveTest");
           msg.setContent("FourthInform");
           send(msg);
           System.out.println("\nFourth INFORM sent");
           finished = true;
           myAgent.doDelete();
          }catch (IOException ioe){
           ioe.printStackTrace();
          }
        }
        public boolean done(){
           return finished;
        }
    });
    }
}
```

File AgentReceiver.java

```
package examples.receivers;

import java.io.*;
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
```

```
    public class AgentReceiver extends Agent {
      class my3StepBehaviour extends SimpleBehaviour {
       final int FIRST = 1;
       final int SECOND = 2;
       final int THIRD = 3;
       private int state = FIRST;
       private boolean finished = false;
        public my3StepBehaviour(Agent a) {
          super(a);
        }
        public void action() {
          switch (state){
           case FIRST: {if (op1())
                         state = SECOND;
                       else
                         state= FIRST;
                       break;}
           case SECOND:{op2(); state = THIRD; break;}
           case THIRD:{op3(); state = FIRST; finished = true; break;}
          }
        }

        public boolean done() {
          return finished;
        }


        private boolean op1(){
         System.out.println( "\nAgent "+getLocalName()+" in state 1.1 is
waiting for a message");
         MessageTemplate m1 =
            MessageTemplate.MatchPerformative(ACLMessage.INFORM);
         MessageTemplate m2 =
            MessageTemplate.MatchLanguage("PlainText");
         MessageTemplate m3 =
            MessageTemplate.MatchOntology("ReceiveTest");
         MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
         MessageTemplate notm3 = MessageTemplate.not(m3);
         MessageTemplate m1andm2_and_notm3 =
            MessageTemplate.and(m1andm2, notm3);

        //The agent waits for a specific message. If it doesn't arrive
        // the behaviour is suspended until a new message arrives.
        ACLMessage msg = receive(m1andm2_and_notm3);
```

```
        if (msg!= null){
          System.out.println("\nAgent "+ getLocalName() +
             " received the following message in state 1.1: " +
             msg.toString());
          return true;
        }
        else {
          System.out.println("\nNo message received in state 1.1");
          block();
          return false;
        }

      }

      private void op2(){
        System.out.println("\nAgent "+ getLocalName() + " in state 1.2
is waiting for a message");
        //Using a blocking receive causes the block
       // of all the behaviours
        ACLMessage msg = blockingReceive(5000);
        if(msg != null)
          System.out.println("\nAgent      "+      getLocalName()      +
             "  received  the  following  message  in  state  1.2:  "
             +msg.toString());
        else
          System.out.println("\nNo message received in state 1.2");


      }

      private void op3() {
        System.out.println("\nAgent: "+getLocalName()+
           " in state 1.3 is waiting for a message");
        MessageTemplate m1 =
           MessageTemplate.MatchPerformative(ACLMessage.INFORM);
        MessageTemplate m2 = MessageTemplate.MatchLanguage("PlainText");
        MessageTemplate m3 =
           MessageTemplate.MatchOntology("ReceiveTest");
        MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
        MessageTemplate m1andm2_and_m3 =
          MessageTemplate.and(m1andm2, m3);
        //blockingReceive and template
        ACLMessage msg = blockingReceive(m1andm2_and_m3);
        if (msg!= null)
          System.out.println("\nAgent      "+      getLocalName()      +
             "  received  the  following  message  in  state  1.3:  "
             + msg.toString());
```

```
        else
          System.out.println("\nNo message received in state 1.3");
        }
      } // End of my3StepBehaviour class


      protected void setup() {
        my3StepBehaviour mybehaviour = new my3StepBehaviour(this);
        addBehaviour(mybehaviour);
      }
    }
```

3.5 Interaction Protocols

FIPA specifies a set of standard interaction protocols, that can be used as standard templates to build agent conversations. For every conversation among agents, JADE distinguishes the *Initiator* role (the agent starting the conversation) and the *Responder* role (the agent engaging in a conversation after being contacted by some other agent). JADE provides ready made behaviour classes for both roles in conversations following most FIPA interaction protocols. These classes can be found in `jade.proto` package, as described in this section.

All Initiator behaviours terminate and are removed from the queue of the agent tasks, as soon as they reach any final state of the interaction protocol. In order to allow the re-use of the Java objects representing these behaviours without having to recreate new objects, all initiators include a number of `reset` methods with the appropriate arguments. Furthermore, all Initiator behaviours, but FipaRequestInitiatorBehaviour, are 1:N, i.e. can handle several responders at the same time.

All Responder behaviours, instead, are cyclic and they are rescheduled as soon as they reach any final state of the interaction protocol. Notice that this feature allows the programmer to limit the maximum number of responder behaviours that the agent should execute in parallel. For instance, the following code ensures that a maximum of two contract-net tasks will be executed simultaneously.

```
Protected void setup() {
   addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
   addBehaviour(new FipaContractNetResponderBehaviour(<arguments>));
}
```

A complete reference for these classes can be found in JADE HTML documentation and class reference.

Since JADE 2.4 a new couple of classes has been added, `AchieveREInitiator/Responder`, that provides an effective implementation for all the FIPA-Request-like interaction protocols, included FIPA-Request itself, FIPA-query, FIPA-propose, FIPA-Request-When, FIPA-recruiting, FIPA-brokering, FIPA-subscribe, ... **It is intention of the authors to keep only this couple of classes and soon deprecate the other jade.proto classes.**

3.5.1 AchieveRE (Achieve **R**ational **E**ffect)

The fundamental view of messages in FIPA ACL is that a message represents a communicative act, just one of the actions that an agent can perform. The FIPA standard specifies for each communicative act the Feasibility Preconditions (the conditions which need to be true

before an agent can execute the action, i.e. before the message can be sent) and the Rational Effect, i.e. the expected effect of the action or, in other terms, the reason why the message is sent. The standard specifies also that, having performed the act (i.e. having sent the message), the sender agent is not entitled to believe that the rational effect necessarily holds; for instance, given its autonomy, the receiver agent might simply decide to ignore the received message. That is not desirable in most applications because it generates an undesirable level of uncertainty. For this reason, instead of sending a single message, an interaction protocol should be initiated by the sender agent that allows to verify if the expected rational effect has been achieved or not.

FIPA has already specified a number of these interaction protocols, like FIPA-Request, FIPA-query, FIPA-propose, FIPA-Request-When, FIPA-recruiting, FIPA-brokering, FIPA-subscribe, that allows the initiator to verify if the expected rational effect of a single communicative act has been achieved. Because they share the same structure, JADE provides the `AchieveREInitiator/Responder` couple of classes which are a single homogeneous implementation of all these kind of interaction protocols.

Figure 5 shows the structure of these interaction protocols. The initiator sends a message (in general it performs a communicative act, as shown in the white box). The responder can then reply by sending a **not-understood**, or a **refuse** to achieve the rational effect of the communicative act, or also an **agree** message to communicate the agreement to perform (possibly in the future) the communicative act, as shown in the first row of shaded boxes. The responder performs the action and, finally, must respond with an **inform** of the result of the action (eventually just that the action has been done) or with a **failure** if anything went wrong. Notice that we have extended the protocol to make optional the transmission of the **agree** message. Infact, in most cases performing the action takes so short time that sending the **agree** message is just an useless and uneffective overhead; in such cases, the **agree** to perform the communicative act is subsumed by the reception of the following message in the protocol.

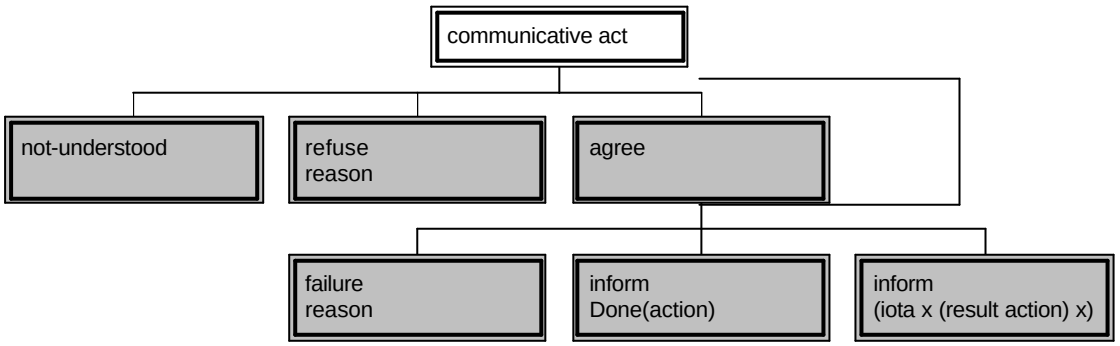


Figure 5 - Homogeneous structure of the interaction protocols.

3.5.1.1 AchieveREInitiator

An instance of this class can be easily constructed by passing, as argument of its constructor, the message used to initiate the protocol. It is important that this message has the right value for the *protocol* slot of the *ACLMessage* as defined by the constants in the interface *FIPAProtocolNames*.

Notice that this *ACLMessage* object might also be incomplete when the constructor of this class is created; the method *prepareRequests* can be overridden in order to return the complete *ACLMessage* or, more exactly (because this initiator allows to manage a 1:N conversation) a Vector of *ACLMessage* objects to be sent.

The class can be easily extended by overriding one (or all) of its *handle...* methods which provide hooks to handle all the states of the protocol. For instance the method *handleRefuse* is called when a *refuse* message is received.

Skilled programmers might find useful, instead of extending this class and overriding some of its methods, registering application-specific *Behaviours* as handler of the states of the protocol, including, for instance, another *AchieveREInitiator* behaviour to request a password before agreeing to perform the communicative act. The methods *registerHandle...* allow to do that. A mix of overridden methods and registered behaviours might often be the best solution.

It is worth clarifying the distinction between the following three handlers:

- *handleOutOfSequence* handles all the unexpected received messages which have the proper conversation-id or in-reply-to value
- *handleAllResponses* handles all the received first responses (i.e. *not-understood, refuse, agree)* and it is called after having called *handleNotUnderstood/Refuse/Agree* for each single response received. In case of 1:N conversations the override of this method might be more useful than the override of the other methods because this one allows to handle all the messages in a single call.
- *handleAllResultNotifications* handles all the received second responses (i.e. *failure, inform)* and it is called after having called *handleFailure/Inform* for each single response received. In case of 1:N conversations the override of this method might be more useful than the override of the other methods because this one allows to handle all the messages in a single call.

A set of variables **(they are not constants!)** is available (*..._KEY*) that provide the keys to retrieve the following information from the dataStore of this Behaviour:

- *getDataStore().get(ALL_RESPONSES_KEY)* returns a *Vector* of *ACLMessage* object with all the first responses (i.e. *not-understood, refuse, agree)*
- *getDataStore().get(ALL_RESULT_NOTIFICATIONS_KEY)* returns a *Vector* of *ACLMessage* object with all the second responses (i.e. *failure, inform)*
- *getDataStore().get(REQUEST_KEY)* returns the *ACLMessage* object passed in the constructor of the class
- *getDataStore().get(ALL_REQUESTS_KEY)* returns the *Vector* of *ACLMessage* objects returned by the *prepareRequests* method. **Remind that** if a Behaviour is registered as handler of the *PrepareRequests* state, it is responsibility of this behaviour to put into the datastore the proper *Vector* of *ACLMessage* objects (bound at the right key) to be sent by this initiator.

This implementation manages the expiration of the timeout, as expressed by the value of the *reply-by* slot of the sent *ACLMessage* objects. In case of 1:N conversation, the minimum is evaluated and used between the values of all the *reply-by* slot of the sent *ACLMessage* objects. Notice that, as defined by FIPA, this timeout refers to the time when the first response (e.g. the *agree* message) has to be received. If applications need to limit the timeout for receiving the last *inform* message, they must embed this limit into the content of the message by using application-specific ontologies.

3.5.1.2 AchieveREResponder

This class is the implementation of the responder role. It is very important to pass the right message template as argument of its constructor, in fact it is used to select which received *ACLMessage* should be served. The method *createMessageTemplate* can be used to create a message template for a given interaction protocol, but also more selective templates might be useful in some cases, for example to have an instance of this class for each possible sender agent.

The class can be easily extended by overriding one (or all) of its *prepare...* methods which provide hooks to handle the states of the protocol and, in particular, to prepare the response messages. The method *prepareResponse* is called when an initiator's message is received and the first response (e.g. the *agree)* must be sent back; the method *prepareResultNotification* is called, instead, when the rational effect must be achieved (for instance the action must be performed in case of a FIPa-Request protocol) and the final response message must be sent back (e.g. the *inform(done)*). **Take care** in returning the proper message and setting all the needed slots of the ACLMessage; in general it is highly recommended to create the reply message by using the method *createReply()* of the class *ACLMessage*.

Skilled programmers might find useful, instead of extending this class and overriding some of its methods, registering application-specific *Behaviours* as handler of the states of the protocol. The methods *registerPrepare...* allow to do that. A mix of overridden methods and registered behaviours might often be the best solution.

A set of variables **(they are not constants!)** is available (*..._KEY*) that provide the keys to retrieve the following information from the dataStore of this Behaviour:

- *getDataStore().get(REQUEST_KEY)* returns the *ACLMessage* object received by the initiator
- *getDataStore().get(RESPONSE_KEY)* returns the first *ACLMessage* object sent to the initiator
- *getDataStore().get(RESULT_NOTIFICATION_KEY)* returns the second *ACLMessage* object sent to the initiator

Remind that if a Behaviour is registered as handler of the *Prepare...* states, it is responsibility of this behaviour to put into the datastore (bound at the right key) the proper *ACLMessage* object to be sent by this responder.

3.5.1.3 Example of using these two generic classes for implementing a specific FIPA protocol

The two classes described above can easily be used for implementing the interaction protocols defined by FIPA.

The following example shows how to add a FIPA-Request initiator behaviour:

```
ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setProtocol(FIPAProtocolNames.FIPA_REQUEST);
request.addReceiver(new AID("receiver", AID.ISLOCALNAME));
myAgent.addBehaviour( new AchieveREInitiator(myAgent, request) {
  protected void handleInform(ACLMessage inform) {
            System.out.println("Protocol finished. Rational Effect achieved.
Received the following message: "+inform);
      }
});
```

The following example shows instead how to add a FIPA-Request responder behaviour:

```
MessageTemplate mt =
      AchieveREResponder.createMessageTemplate(FIPAProtocolNames.FIPAREQUEST);
myAgent.addBehaviour( new AchieveREResponder(myAgent, mt) {
  protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage
response) {
      System.out.println("Responder has received the following message: " +
request);
      ACLMessage informDone = request.createReply();
      informDone.setPerformative(ACLMessage.INFORM);
```

```
        informDone.setContent("inform done");
        return informDone;
        }
});
```

3.5.2 FIPA-Contract-Net

This interaction protocol allows the Initiator to send a Call for Proposal to a set of responders, evaluate their proposals and then accept the preferred one (or even reject all of them). The interaction protocol is deeply described in the FIPA specifications while the following figure is just a simplification for the programmer.

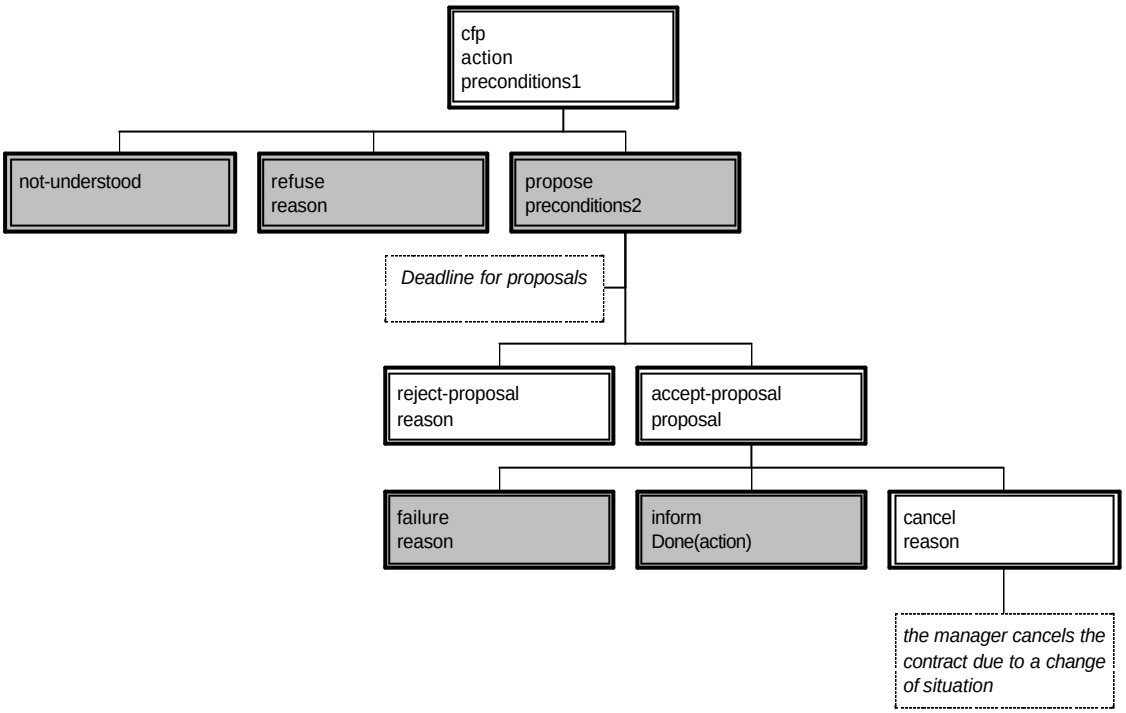


Figure 6 - FIPA-Contract-Net Interaction Protocol

3.5.2.1 FipaContractNetInitiatorBehaviour

This abstract behaviour implements the fipa-contract-net interaction protocol from the point of view of the agent initiating the protocol, that is the agent that sends the cfp (call for proposal) message.

The constructor of this behaviour takes 3 parameters
`public FipaContractNetInitiatorBehaviour(Agent a, ACLMessage msg, List responders)`
the calling agent, the CFP message to be sent and the group of agents to which the message should be sent. In fact, the protocol is implemented 1:N with one initiator and several responders.

The programmer should implement the two methods `handleProposeMessages` and `handleFinalMessages` to handle the two states of the protocol from the point of view of the initiator.

Under some circumstances, for instance when using the SL-0 content language, the content of the CFP message needs to be adapted to each receiver. For this reason, the method `createcfpcontent` is called before sending each message. The default implementation returns exactly the same content independently of the receiver; the programmer might also wish to override this default implementation.

The behaviour takes also care of handling timeouts in waiting for the answers. The timeout is got from the `reply-by` field of the ACLMessage passed in the constructor; if it was not set, then an infinite timeout is used. If the timeout expires without having received any answer, the method `handleXXXMessages` is executed by passing an empty vector of messages. Of course, late answers that arrive after the timeout expires are not consumed and remain in the private queue of incoming ACLmessages. Because this queue has a maximum size, these messages will be removed after the queue becomes full.

3.5.3 FipaContractNetResponderBehaviour

This abstract behaviour class implements the fipa-contract-net interaction protocol from the point of view of a responder to a call for proposal (cfp) message.

The programmer should extend this class by implementing the `handleXXX` methods that are called to handle the types of messages that can be received in this protocol.

3.5.4 Generic states of interaction protocols

The package jade.proto.states contains implementations for some generic states of interaction protocols which might be useful to register as handlers.

3.5.4.1 HandlerSelector class

This abstract class of the package jade.proto.states provides an implementation for a generic selector of handler, where an handler is a jade.core.behaviours.Behaviour.

The constructor of the class requires passing three arguments: a reference to the Agent, a reference to the DataStore where the selection variable can be retrieved, and, finally, the access key to retrieve the selection variable from this DataStore.

This selection variable will be later passed as argument to the method *getSelectionKey* that must return the key for selecting between the registered handlers. In fact, each handler must be registered with a key via the method *registerHandler*.

Useful examples of usage of this class are, for instance, the selection of a different handler for each action name (es. the action "register" is handled by the behaviour "registerBehaviour", the action modify by another one, and so on for each action). This class is generic enough to allow a large variety of selection systems, such as based on the message sender, the content language, the ontology, ... the programmer just needs to extend the class and override its method `getSelectionKey`

3.5.4.2 MsgReceiver class

This is a generic implementation for waiting for the arrival of a given message of the expiration of a given timeout. Refer to the javadoc for the documentation of its usage.

3.6 Application-defined content languages and ontologies

3.6.1 Rationale

When an agent A communicates with another agent B, a certain amount of information I is transferred from A to B by means of an ACL message.

Inside the ACL message I is represented as a content expression consistent with a proper content language (e.g. SL) and encoded in a proper format (e.g. string).

Both A and B have their own (possibly different) way of internally representing I.

Taking into account that the way an agent internally represents a piece information must allow an easy handling of that piece of information, it is quite clear that the representation used in an ACL content expression is not suitable for the inside of an agent.

For example the information that *the person Giovanni is 33 years old* in an ACL content expression could be represented as the string

```
(person (name Giovanni) (age 33) )
```

Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Giovanni would require each time to parse the string.

Considering software agents written in Java (as JADE agents are), information can conveniently be represented inside an agent as Java objects.

For example representing the above information about Giovanni as an instance (an object) of an application-specific class

```
class Person {
  String name;
  int age;

  public String getName() {return name; }
  public void setName(String n) {name = n; }
  public int getAge() {return age; }
  public void setAge(int a) {age = a; }
  ….
}
```

initialized with

```
name = "Giovanni";
age = 33;
```

would allow to handle it very easily.

It is clear however that if on the one hand information handling inside an agent is eased, on the other hand each time agent A sends a piece of information I to agent B,

1) A needs to convert his internal representation of I into the corresponding ACL content expression representation and B needs to perform the opposite conversion.
2) Moreover B should also check that I complies with the rules (i.e. for instance that the age of Giovanni is actually an integer value) of the ontology by means of which both A and B ascribe a proper meaning to I.

The support for application-defined ontology and content languages provided by JADE is designed to support agent internal representation of information as Java objects, as described above, by minimizing the developer effort in performing the above conversion and check operations.

3.6.2 The conversion pipeline

Each time an information has to be inserted into or extracted from an ACL content expression the JADE framework automatically performs the pipeline depicted in figure.

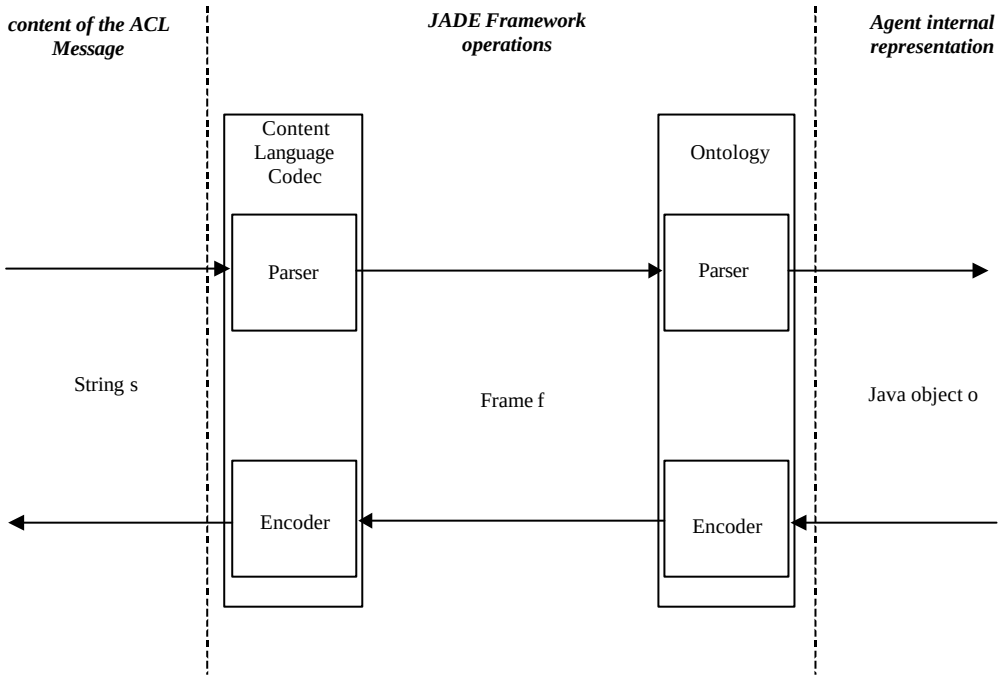


Figure 7 - Pipeline of the message content encoding/decoding

First an appropriate content language **codec** object is able to parse a content expression and to convert it into a t-uple[7] of `Frame`[8] objects.

An appropriate **ontology** object is then able to check whether a `Frame` object is consistent with one of the schemas defining the **roles**[9] included in the ontology and, in this case, to convert

[7] A content expression can include more than one entity in the domain of discourse. E.g. the content of a REFUSE ACL message is a t-uple with 2 elements: an action expression and the reason why the agent sending the message refuses to accomplish that action.

[8] The JADE Framework uses an internal representation of information based on the class `Frame`. A `Frame` object has a name and a set of slots each one being characterized by a name, a position (within the frame) and an untyped value. Whatever entity in the domain of discourse (i.e. whatever information) can be represented as a `Frame` object.

[9] An ontology basically includes all the concepts, predicates and actions, collectively called **roles**, that are meaningful for the agents sharing this ontology. For instance the concepts *Company* and *Person*, the predicate *WorfsFor* and the action *Engage* can be **roles** in an ontology dealing with employees. All elements in the domain of discourse (e.g. the person Giovanni) are instances of one of the **roles** composing the ontology.

the `Frame` object into a properly initialized instance of the application-specific class (e.g. the Person class mentioned above) representing the matched role.

The opposite pipeline allows to convert a sentence belonging to the domain of discourse, and represented as a Java object, into the appropriate content language and encoding.

The JADE framework hides the stages of this pipeline to the programmer who just needs to call the following methods of the `Agent` class.

```
List extractContent(ACLMessage msg);
void fillContent(ACLMessage msg, List content);
```

As already mentioned the content of an ACL message is in general a t-uple of entity inn the domain of discourse. In Java this is represented as a `List`.

The programmer however has to create and add to the resources of the agent the codec and ontology objects mentioned above as described in the followings.

3.6.3 Codec of a Content Language

Each content language codec in JADE must implement the interface `jade.lang.Codec` and, in particular, the two methods `decode`() and `encode`() to respectively

- parse the content in an ACL message and convert it into a `List` of `Frame` objects.
- encode the content from a `List` of `Frame` objects into the content language syntax and encoding.

The `Frame` class is a neutral type (i.e. it does not distinguish between concepts, actions and predicates), that has been designed in order to allow accessing its slots both by name (e.g. *(divide :dividend 10 :divisor 2)*) and by position (e.g. *(divide 10 2)*).

This `Codec` object must then be added to the resources of each agent, which wishes to use that language, by using the method `registerLanguage`() available in the `Agent` class.

By means of this operation a `Codec` object is associated to a content language name. When the `fillContent`() and `extractContent`() methods are called the `Codec` object associated to the content language indicated in the `:language` slot of the ACL message will be used to perform the conversion pipeline described in previous chapter.

Notice that JADE already includes the `Codec` for SL-0 (one of the standard content languages defined by FIPA) that is the class `jade.lang.sl.SL0Codec`. For an agent using SL0 it will be therefore sufficient to insert the instruction

```
registerLanguage("SL0", new SL0Codec());
```

3.6.4 Creating an Ontology

Each ontology in JADE must implement the `jade.onto.Ontology` interface.

It is important to note however that in the adopted approach an ontology is represented by an instance of a class implementing the `jade.onto.Ontology` interface and not just by that class.

More in detail a class implementing the `jade.onto.Ontology` interface only embeds the definition of the semantic checks that will be performed when some information is received. For example an implementation can check that the age of a person is an integer value, while another implementation can also check that that integer value is > 0. All the ontological roles included in the ontology (such as the concept of person) must on the other hand be added at run-time to an instance of the above class.

Two instances `o1` and `o2` of the same class `O` implementing the `jade.onto.Ontology` interface can represent two different ontologies provided that at run-time different ontological roles are added to `o1` and `o2`.

A class, `jade.onto.DefaultOntology`, providing a default implementation of the `Ontology` interface is already provided by JADE. This is simple but still expected to be useful in most practical applications.

Creating an ontology requires the following steps:

- Defining an application-specific class for each role in the ontology
- Creating an object of class `DefaultOntology`
- Adding to that object all the ontological roles as described below.

Each ontological role is described by a name and a number of **slots**. The `SlotDescriptor` class is provided to describe the characteristics of a slot of an ontological role.

The method `addRole()` by means of which a role is added to an ontology object takes therefore the following parameters:

- A `String` indicating the name of the added role. This parameter is missing for an unnamed slot.
- An array of `SlotDescriptor` each one describing a slot of the added role.
- The Java class, if any, that represents the role.

For example, adding the *person* role described by the `Person` class mentioned above, to a previously created ontology object `myOnto` will look like

```
Ontology myOnto = new DefaultOntology();
……
myOnto.addRole(
  “Person”,
  new SlotDescriptor[]{
        new SlotDescriptor(“name”, Ontology.PRIMITIVE_SLOT,
        Ontology.STRING_TYPE, Ontology.M),
        new SlotDescriptor(“age”, Ontology.PRIMITIVE_SLOT,
        Ontology.INTEGER_TYPE,Ontology.O)
  },
  Person.class
);
```

Each slot has

- A **name** and/or a **position** (implicitly defined by the position in the array of `SlotDescriptors`) identifying the slot.
- A **category** stating that the value of the slot can be a primitive entity such as a string or an integer (`Ontology.PRIMITIVE_SLOT`), an instance of another ontological role (`Ontology.FRAME_SLOT`) or a set (`Ontology.SET_SLOT`) or sequence (`Ontology.SEQUENCE_SLOT`) of entities.
- A **type** defining the primitive type (for primitive slots) or role (for frame slots) of the value of the slot or of the elements in the set/sequence in case of set slots or sequence slots.
- A **presence** flag defining whether the slot is mandatory (`Ontology.M`) or optional (`Ontology.O`).

In the above case the *person* role has two named slots called *name* and *age*. The first is mandatory (an exception will be thrown if this slot has a null value) and permitted values are of type String. The second is optional and permitted values are of type Integer.

As a further example three other roles are added to the ontology represented by the `myOnto` object.

- *Address*, with three named slots, *street*, *number* and *city*, of type `String`, `Integer` and `String` respectively and all mandatory.
- *Company* with two named slots, *name* and *address*, of type `String` and *Address* (i.e. the values of this slot are instances of the *Address* role) respectively, one mandatory and the other optional.
- *Engage* (the action of engaging a person in a company) with two unnamed slots of type *Person* and *Company* respectively and both mandatory.

Application specific class representing the *Address* role

```
public class Address {
  private String street;
  private Integer number;
  private String city;
  public String getStreet() { return string; }
  public void setStreet(String s) { string = s; }
  public Integer getNumber() { return number; }
  public void setNumber(Integer n) { number = n; }
  public String getCity() { return city; }
  public void setCity(String c) { city  = c; }
}
```

Application specific class representing the *Company* role

```
public class Company {
    private String name;
    private Address  address;

    public void   setName(String n) { name = n; }
    public String getName() { return name; }
    public void   setAddress(Address a) { address = a; }
    public Address  getAddress() { return address; }
}
```

Application specific class representing the *Engage* role

```
public class Engage {
    private Person personToEngage;
    private Company engager;

    public void   set_0(Person p) { personToEngage = p; }
    public Person  get_0() { return personToEngage; }
    public void   set_1(Company c) { engager = c; }
    public Company  get_1() { return engager; }
}
```

Code for adding the *Address*, *Company* and *Engage* roles to the ontology.

```
myOnto.addRole(
  "Address",
  new SlotDescriptor[]{
       new SlotDescriptor("street", Ontology.PRIMITIVE_SLOT,
       Ontology.STRING_TYPE, Ontology.M),
       new SlotDescriptor("number", Ontology.PRIMITIVE_SLOT,
       Ontology.INTEGER_TYPE,Ontology.M)
       new SlotDescriptor("city", Ontology.PRIMITIVE_SLOT,
       Ontology.SRING_TYPE,Ontology.M)
  },
  Address.class
);

myOnto.addRole(
  "Company",
  new SlotDescriptor[]{
       new SlotDescriptor("name", Ontology.PRIMITIVE_SLOT,
       Ontology.STRING_TYPE, Ontology.M),
       new SlotDescriptor("address", Ontology.FRAME_SLOT,
       "Address" ,Ontology.O)
  },
  Company.class
);

myOnto.addRole(
  "engage",
  new SlotDescriptor[]{
       new SlotDescriptor(Ontology.FRAME_SLOT, "Person"
       Ontology.M),
       new SlotDescriptor(Ontology.FRAME_SLOT, "Company",
       Ontology.M)
  },
  Engage.class;
```

```
);
```

The ontology object must finally be added to the resources of each agent wishing to use it, by using the method `registerOntology()` available in the `Agent` class.

By means of this operation an `Ontology` object is associated to a name. When the `fillContent()` and `extractContent()` methods are called the `Ontology` object associated to the content language indicated in the `:ontology` slot of the ACL message will be used to perform the conversion pipeline described in section 3.6.2.

3.6.5 Application specific classes representing ontological roles

In order to represent an ontological role (i.e. in order to be accepted by the `Ontology` object), a Java class must obey to some rules:

1) For each slot in the represented role named *XXX*, of category `Ontology.PRIMITIVE_SLOT` or `Ontology.FRAME_SLOT` and of type `T` the class must have two accessible methods with the following signature:

```
public T getXXX();
public void setXXX(T t);
```

2) For each slot in the represented role named *XXX*, of category `Ontology.SET_SLOT` or `Ontology.SEQUENCE_SLOT` and with elements of type `T`, the class must have two accessible methods with the following signature:

```
public Iterator getAllXXX();
public void addXXX(T t);
```

3) For each unnamed slot use "_p" (being p the position of the slot) instead of the slot name for the get and set methods (see the Engage class mentioned above for an example).

4) In all previous cases the type `T` cannot be a primitive type such as `int`, `float` or `boolean`. Use `Integer`, `Float`, `Boolean` …. instead.

3.6.6 Discovering the ontological role of a Java object representing an entity in the domain of discourse

As already mentioned, when an ACL message is received, provided that the proper ontology and content language codec objects has been previously registered, the content of the ACL message can be easily converted into a list of proper Java objects by means of the `extractContent()` method .

```
List l = extractContent( msg );
```

In general however the receiving agent does not know a-priori the role of each Java object in the list. In order to discover it the ontology object must be used as described in the example below refering to the first object in the list.

```
Object obj = l.get(0);
Ontology onto = lookupOntology(msg.getOntology());
String roleName = onto.getRoleName(obj.getClass());
```

The `lookupOntology`() is a method of the `Agent` class that returns the ontology object previously associated to a given name by calling the `registerOntology`() method.

Once discovered the role of the entity represented by an object it will be possible to cast it to the application specific class representing that role.

3.6.7 Setting and getting the content of an ACL message.

Having registered a content language codec and an ontology with the agent, it is possible to exploit the automatic support of the JADE framework to set and get the content of an ACL message. The `Agent` class provides two methods for this purpose: `extractContent()` and `fillContent()` to implement parsing and encoding operations on the message content, respectively.

The first method extracts the content from an ACL message and returns a `List` of Java objects (one object for each element of the tuple in the content) by calling the appropriate content language `Codec` (according to the value of the *:language* parameter of the ACL message) and the appropriate Ontology (according to the value of the *:ontology* parameter of the ACL message).

The second method, instead, makes the opposite operation, that is it fills in the content of an ACL message by interpreting a `List` of Java objects with the appropriate `Ontology` and content language `Codec`, as specified by the values of the *:ontology* and the *:language* parameter of the ACL message.

Refer to the javadoc documentation for a detailed description of the usage of these two methods.

3.7 Support for Agent Mobility

Using JADE, application developers can build mobile agents, which are able to migrate or copy themselves across multiple network hosts. In this version of JADE, only *intra-platform* mobility is supported, that is a JADE mobile agent can navigate across different agent containers but it is confined to a single JADE platform.

Moving or cloning is considered a state transition in the life cycle of the agent. Just like all the other life cycle operation, agent motion or cloning can be initiated either by the agent itself or by the AMS. The `Agent` class provides a suitable API, whereas the AMS agent can be accessed via FIPA ACL as usual.

Mobile agents need to be *location aware* in order to decide when and where to move. Therefore, JADE provides a proprietary ontology, named *jade-mobility-ontology*, holding the necessary concepts and actions.

This ontology is contained within the `jade.domain.MobilityOntology` class, and it is an example of the new application-defined ontology support.

3.7.1 JADE API for agent mobility.

The two public methods `doMove()` and `doClone()` of the `Agent` class allow a JADE agent to migrate elsewhere or to spawn a remote copy of itself under a different name. Method `doMove()` takes a `jade.core.Location` as its single parameter, which represents the intended destination for the migrating agent. Method `doClone()` also takes a `jade.core.Location` as parameter, but adds a `String` containing the name of the new agent that will be created as a copy of the current one.

Looking at the documentation, one finds that `jade.core.Location` is an abstract interface, so application agents are not allowed to create their own locations. Instead, they must ask the AMS for the list of the available locations and choose one. Alternatively, a JADE agent can also request the AMS to tell where (at which location) another agent lives.

Moving an agent involves sending its code and state through a network channel, so user defined mobile agents must manage the serialization and unserialization process. Some among the various resources used by the mobile agent will be moved along, while some others will be disconnected before moving and reconnected at the destination (this is the same distinction between `transient` and non-`transient` fields used in the *Java Serialization API*). JADE makes available a couple of matching methods in the `Agent` class for resource management.

For agent migration, the `beforeMove()` method is called at the starting location just before sending the agent through the network (with the scheduler of behaviours already stopped), whereas the `afterMove()` method is called at the destination location as soon as the agent has arrived and its identity is in place (but the scheduler has not restarted yet).

For agent cloning, JADE supports a corresponding method pair, the `beforeClone()` and `afterClone()` methods, called in the same fashion as the `beforeMove()` and `afterMove()` above. The four methods above are all `protected` methods of the `Agent` class, defined as empty placeholders. User defined mobile agents will override the four methods as needed.

3.7.2 JADE Mobility Ontology.

The *jade-mobility-ontology* ontology contains all the concepts and actions needed to support agent mobility. JADE provides the class `jade.domain.MobilityOntology`, working as a *Singleton* and giving access to a single, shared instance of the JADE mobility ontology through the `instance()` method.

The ontology contains ten frames (six concepts and four actions), and a suitable inner class is associated with each frame using a `RoleEntityFactory` object (see Section 3.6.4 for details). The following list shows all the frames and their structure.

- `Mobile-agent-description`; describes a mobile agent going somewhere. It is represented by the `MobilityOntology.MobileAgentDescription` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**AID**	**Mandatory**
destination	**Location**	**Mandatory**
agent-profile	mobile-agent-profile	Optional
agent-version	String	Optional
signature	String	Optional

- `mobile-agent-profile`; describes the computing environment needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentProfile` inner class.

Slot Name	Slot Type	Mandatory/Optional
system	mobile-agent-system	Optional
language	mobile-agent-language	Optional
os	**Mobile-agent-os**	**Mandatory**

- `mobile-agent-system`; describes the runtime system used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentSystem` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `mobile-agent-language`; describes the programming language used by the mobile agent. It is represented by the `MobilityOntology.MobileAgentLanguage` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `mobile-agent-os`; describes the operating system needed by the mobile agent. It is represented by the `MobilityOntology.MobileAgentOS` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
major-version	**Long**	**Mandatory**
minor-version	Long	Optional
dependencies	String	Optional

- `Location`; describes a location where an agent can go. It is represented by the `MobilityOntology.Location` inner class.

Slot Name	Slot Type	Mandatory/Optional
name	**String**	**Mandatory**
protocol	**String**	**Mandatory**

address	String	Mandatory

- `move-agent`; the action of moving an agent from a location to another. It is represented by the `MobilityOntology.MoveAction` inner class.

  This action has a single, unnamed slot of type `mobile-agent-description`. The argument is mandatory.

- `clone-agent`; the action performing a copy of an agent, possibly running on another location. It is represented by the `MobilityOntology.CloneAction` inner class.

  This action has two unnamed slots: the first one is of `mobile-agent-description` type and the second one is of `String` type. Both arguments are mandatory.

- `where-is-agent`; the action of requesting the location where a given agent is running. It is represented by the `MobilityOntology.WhereIsAgent` inner class.

  This action has a single, unnamed slot of type `AID`. The argument is mandatory.

- `query-platform-locations`; the action of requesting the list of all the platform locations. It is represented by the `MobilityOntology.QueryPlatformLocations` inner class.

This action has no slots.

Notice that this ontology has no counter-part in any FIPA specifications. It is intention of the JADE team to update the ontology as soon as a suitable FIPA specification will be available.

3.7.3 Accessing the AMS for agent mobility.

The JADE AMS has some extensions that support the agent mobility, and it is capable of performing all the four actions present in the *jade-mobility-ontology*. Every mobility related action can be requested to the AMS through a *FIPA-request* protocol, with *jade-mobility-ontology* as `ontology` value and *FIPA-SL0* as `language` value.

The `move-agent` action takes a `mobile-agent-description` as its parameter. This action moves the agent identified by the `name` and `address` slots of the `mobile-agent-description` to the location present in the `destination` slot.

For example, if an agent wants to move the agent *Peter* to the location called *Front-End*, it must send to the AMS the following ACL `request` message:

```
(REQUEST
 :sender (agent-identifier :name RMA@Zadig:1099/JADE)
 :receiver (set (agent-identifier :name ams@Zadig:1099/JADE))
 :content  (
   (action (agent-identifier :name ams@Zadig:1099/JADE)
           (move-agent (mobile-agent-description
```

```
              :name (agent-identifier :name Johnny@Zadig:1099/JADE)
              :destination (location
                :name Main-Container
                :protocol JADE-IPMT
                :address Zadig:1099/JADE.Main-Container )
              )
             )
    )
  )
  :reply-with  Req976983289310
  :language  FIPA-SL0
  :ontology  jade-mobility-ontology
  :protocol  fipa-request
  :conversation-id  Req976983289310
 )
```

The above message was captured using the JADE sniffer, using the *MobileAgent* example and the RMA support for moving and cloning agents.

Using JADE ontology support, an agent can easily add mobility to its capabilities, without having to compose ACL messages by hand.

First of all, the agent has to create a new `MobilityOntology.MoveAction` object, fill its argument with a suitable `MobilityOntology.MobileAgentDescription` object, filled in turn with the name and address of the agent to move (either itself or another mobile agent) and with the `MobilityOntology.Location` object for the destination. Then, a single call to the `Agent.fillContent()` method can turn the `MoveAction` Java object into a `String` and write it into the `content` slot of a suitable `request` ACL message.

The `clone-agent` action works in the same way, but has an additional `String` argument to hold the name of the new agent resulting from the cloning process.

The `where-is-agent` action has a single `AID` argument, holding the identifier of the agent to locate. This action has a result, namely the location for the agent, that is put into the `content` slot of the `inform` ACL message that successfully closes the protocol.

For example, the `request` message to ask for the location where the agent *Peter* resides would be:

```
 (REQUEST
   :sender   (agent-identifier :name da1@Zadig:1099/JADE)
   :receiver (set (agent-identifier :name ams@Zadig:1099/JADE))
   :content  (( action
     (agent-identifier :name ams@Zadig:1099/JADE)
     (where-is-agent (agent-identifier :name Peter@Zadig:1099/JADE))
   ))
   :language  FIPA-SL0
   :ontology  jade-mobility-ontology
   :protocol  fipa-request
 )
```

The resulting `Location` would be contained within an `inform` message like the following:

```
(INFORM
  :sender (agent-identifier :name ams@Zadig:1099/JADE)
  :receiver  (set (agent-identifier :name da1@Zadig:1099/JADE))
  :content  ((result
    (action
      (agent-identifier :name ams@Zadig:1099/JADE)
      (where-is-agent (agent-identifier :name Peter@Zadig:1099/JADE))
    )
    (set (location
      :name Container-1
      :protocol JADE-IPMT
      :address Zadig:1099/JADE.Container-1
    ))
  ))
  :reply-with  da1@Zadig:1099/JADE976984777740
  :language  FIPA-SL0
  :ontology  jade-mobility-ontology
  :protocol  fipa-request
)
```

The `query-platform-locations` action takes no arguments, but its result is a set of all the `Location` objects available in the current JADE platform. The message for this action is very simple:

```
( REQUEST
 :sender (agent-identifier :name Johnny)
 :receiver (set (Agent-Identifier :name AMS))
 :content (( action (agent-identifier :name AMS)
            ( query-platform-locations ) ))
 :language FIPA-SL0
 :ontology jade-mobility-ontology
 :protocol fipa-request
)
```

If the current platform had three containers, the AMS would send back the following `inform` message:

```
( INFORM
 :sender (Agent-Identifier :name AMS)
 :receiver (set (Agent-Identifier :name Johnny))
 :content (( Result ( action (agent-identifier :name AMS)
                             ( query-platform-locations ) )
            (set (Location
                  :name Container-1
                  :transport-protocol JADE-IPMT
```

```
                    :transport-address IOR:000….Container-1 )
                  (Location
                    :name Container-2
                    :protocol JADE-IPMT
                    :address IOR:000….Container-2 )
                 (Location
                    :name Container-3
                    :protocol JADE-IPMT
                    :address IOR:000….Container-3 )
            )))
   :language  FIPA-SL0
   :ontology  jade-mobility-ontology
   :protocol  fipa-request
  )
```

The `MobilityOntology.Location` class implements `jade.core.Location` interface, so that it can be passed to `Agent.doMove()` and `Agent.doClone()` methods. A typical behaviour pattern for a JADE mobile agent will be to ask the AMS for locations (either the complete list or through one or more `where-is-agent` actions); then the agent will be able to decide if, where and when to migrate.

3.8 Using JADE from external Java applications

Since JADE 2.3, an in-process interface has been implemented that allows external Java applications to use JADE as a kind of library and to launch the JADE Runtime from within the application itself.

A singleton instance of the JADE Runtime can be obtained via the static method *jade.core.Runtime.instance()* Then, it provides two methods to create a JADE main-container or a JADE remote container (i.e. a container that joins to an existing main-container forming in this way a distributed agent platform); both methods requires passing as a parameter an object that implements the *jade.core.Profile* interface that can be queried, via the *getParameter(name)* method, to get the hostname and port number of the main container.

Both these two methods of the Runtime return a wrapper object, belonging to the package *jade.wrapper*, that wraps the higher-level functionality of the agent containers, such as installing and uninstalling MTPs (Message Transport Protocol)[10], killing the container (where just the container is killed while the external application remains alive) and, of course, creating new agents. The *createAgent* method of this container wrapper returns as well a wrapper object, which wraps some functionalities of the agent, but still tends to preserve the autonomy of agents. In particular, the application can control the life-cycle of the Agent but it cannot obtain a direct reference to the Agent object and, as a direct consequence, it cannot perform method calls on that object. **Notice that**, having created the agent, it still needs to be started via the method *start()*

The following code lists a very simple way to launch an agent from within an external applications (refer also to the *inprocess* directory in the JADE *examples* that contains an example of usage of this wrapping and in-process interface).

[10] see also the Administrator's Guide for this functionality

```
import jade.core.Runtime;
import jade.core.Profile;
import jade.core.ProfileImpl;
import jade.wrapper.*;
...
// Get a hold on JADE runtime
Runtime rt = Runtime.instance();
// Create a default profile
Profile p = new ProfileImpl();
// Create a new non-main container, connecting to the default
// main container (i.e. on this host, port 1099)
AgentContainer ac = rt.createAgentContainer(p);
// Create a new agent, a DummyAgent
// and pass it a reference to an Object
Object reference = new Object();
Object args[] = new Object[1];
args[0]=reference;
Agent dummy = ac.createAgent("inProcess",
              "jade.tools.DummyAgent.DummyAgent", reference);
// Fire up the agent
dummy.start();
...
```

Notice that this mechanism allows several different configurations for a JADE platform, such as a complete in-process platform composed of several containers on the same JVM, a platform partly in-process (i.e. containers launched by an external Java application) and partly out-of-process (i.e. containers launched from the command line).

4 A SAMPLE AGENT SYSTEM

We are presenting an example of an agent system explaining how to use the features available in JADE framework. In particular we will show the possibility of organising the behaviour of a single agent in different sub-behaviours and how the message exchange among agents takes place.

The agent system, in the example, is made of two agents communicating through FIPA request protocol.

This section is still to do. Please refer to JADE examples present in src/examples directory. Refer also to the `README` *file in* `src/examples` *directory to get some explanations of each example program.*

5 APPENDIX A: CONTENT-LANGUAGE INDEPENDENT API
FEDERICO BERGENTI (UNIVERSITY OF PARMA)

Application-specific ontologies describe the elements that agents use to create the content of messages, e.g., application-specific predicates and actions. The package `jade.content` (and its sub-packages) allows to create application-specific ontologies and to use them independently of the adopted content language: the code that implements the ontology and the code that sends and receives messages do not depend on the content language. The following is a description of such a package that uses `src/example/content` as a running example.

5.1 Creating an Application-Specific Ontology

An ontology defines a vocabulary and a set of relationships between the elements of the vocabulary. The relationships can be:

1) structural, e.g., the predicate `fatherOf` is defined over two parameters, a father and a set of children because we want to use it to say `fatherOf(John, (Mary, Lisa))`;
2) semantic, e.g., a concept belonging to the class `Man` also belongs to the class `Person`.

An application-specific ontology is implemented through one object of class `FullOntology` and it is characterized by:

1) one name;
2) one base ontology at most, i.e., an ontology that it extends;
3) a vocabulary;
4) a set of element schemata.

The following code implements the `People` ontology: it defines a constant for each element (concept, action, predicate, etc.) that we want to include in the vocabulary.

```
public class PeopleOntology extends FullOntology {
  // The name of this ontology.
  public static final String ONTOLOGY_NAME = "PEOPLE_ONTOLOGY";

  // Concepts, i.e., objects of the world.
  public static final String PERSON  = "PERSON";
  public static final String MAN     = "MAN";
  public static final String WOMAN   = "WOMAN";
  public static final String ADDRESS = "ADDRESS";

  // Slots of concepts, i.e., attributes of objects.
  public static final String NAME   = "NAME";
  public static final String STREET = "STREET";
  public static final String NUMBER = "NUMBER";
  public static final String CITY   = "CITY";

  // Predicates
  public static final String FATHER_OF = "FATHER_OF";
  public static final String MOTHER_OF = "MOTHER_OF";
```

```
  // Roles in predicates, i.e., names of arguments for predicates
  public static final String FATHER   = "FATHER";
  public static final String MOTHER   = "MOTHER";
  public static final String CHILDREN = "CHILDREN";

  // Actions
  public static final String MARRY = "MARRY";

  // Arguments in actions
  public static final String HUSBAND = "HUSBAND";
  public static final String WIFE    = "WIFE";

  private static PeopleOntology theInstance = new PeopleOntology();

  public static PeopleOntology getInstance() {
    return theInstance;
  }

  public PeopleOntology(Ontology base) {
    super(ONTOLOGY_NAME, ACLOntology.getInstance());

    // Add definitions of schemata here.
    …
  }
}
```

The constructor calls `super()` to assign a name to the ontology and to declare that it extends the `ACLOntology`. `People` ontology, and reasonably all ontologies, extends `jade.content.onto.ACLOntology` because we want to use in our messages the elements of such an ontology, e.g., variables and the `Done` predicate. If you do not need ACL concepts, you can extend `jade.content.onto.BasicOntology` in order to have only basic types, i.e., lists, strings and numbers. `ACLOntology` extends the `BasicOntology`.

The definition of the ontology in the example is not complete, we have to substitute dots with the definition of the element schemata. Element schemata are objects describing the structure of concepts, actions, predicate, etc. that we allow in our messages. In the `People` ontology they describe what a person is, what an address is, what a father is, etc. The following is the element schema for the concept of `Person`. This schema states that a `Person` is characterized by a `name` and an `address`:

```
// Get the element schema for strings from BasicOntology
PrimitiveSchema stringSchema =
  (PrimitiveSchema)getSchema(BasicOntology.STRING);

// Define the concept of Person
ConceptSchema personSchema = new ConceptSchema(PERSON);
personSchema.add(NAME,    stringSchema);
```

```
personSchema.add(ADDRESS, addressSchema, ObjectSchema.OPTIONAL);

// Add the schema to the ontology
add(personSchema);
```

PERSON, NAME and ADDRESS are string defined in the vocabulary and addressSchema has been defined before (not shown). Schemata that describe concepts support inheritance[11]. You can define the concept of Man as a refinement of the concept of Person:

```
ConceptSchema manSchema = new ConceptSchema(MAN);
manSchema.addSuperSchema(personSchema);
```

Element schema describe, in some way, the structure of a class of objects and therefore they can be associated with Java classes. This maps elements of the ontology that comply with a schema with Java objects of that class. The following is a class that might be associated with the Person concept:

```
public class Person extends Concept {
  private String  name    = null;
  private Address address = null;

  public void setName(String name) {
    this.name = name;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

  public String getName() {
    return name;
  }

  public Address getAddress() {
    return address;
  }
}
```

The association between the class and the schema is performed when registering the concept in the ontology using the following statement:

[11] Only concept schemata support inheritance, all other schemata, e.g., predicate schemata, action schemata, etc do not.

```
// Add the schema to the ontology
addElement(personSchema, Person.class);
```

Associating classes with schemata is not mandatory, but helps because it support easier APIs, as shown later. In order to associate a class with a schema, the class must:

1) extend a class in `jade.content`, e.g., `Person` extends `Concept` because we want to associate it with a concept schema;
2) provide public `get`/`set` methods for each attribute (you can use basic types like `int` or `boolean`);
3) provide a constructor with no parameters, i.e., the default contructor.

Defining actions, predicates, etc. is just like defining concepts. Figure 8 shows the classes that corresponds to the elements we can use in our ontologies.

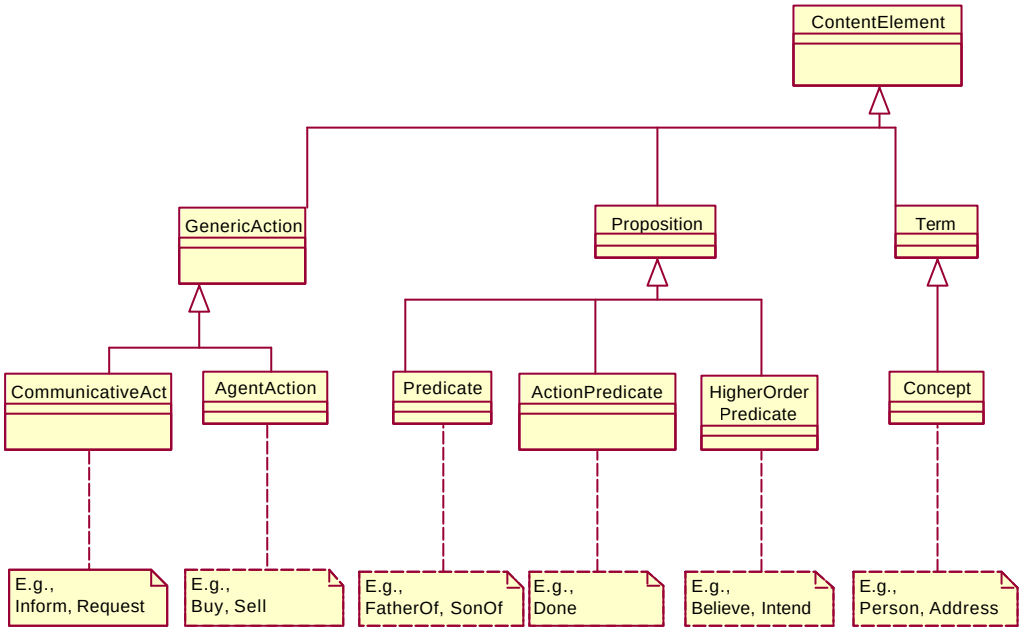


Figure 8 - Classes that correspond to elements of ontologies.

The following is the definition of the predicate `fatherOf`:

```
// Define a schema for the set of children
AggregateSchema childrenSchema = new AggregateSchema(BasicOntology.SET);

// Define the schema for fatherOf predicate
PredicateSchema fatherOfSchema = new PredicateSchema(FATHER_OF);

fatherOfSchema.add(FATHER,   manSchema);
fatherOfSchema.add(CHILDREN, childrenSchema);
```

```
// Add the predicate to the ontology
add(fatherOfSchema, FatherOf.class);
```

First we define a new schema to describe the set of children. Then we define the predicate schema for `fatherOf` by introducing two roles: the `father` and the `children`. Finally we register `fatherOfSchema` with the ontology associating it with the following class:

```
public class FatherOf extends Predicate {
  private List children = null;
  private Man  father   = null;

  public void setChildren(List children) {
    this.children = children;
  }

  public void setFather(Man father) {
    this.father = father;
  }

  public Man getFather() {
    return father;
  }

  public List getChildren() {
    return children;
  }
}
```

Note that we use `jade.util.leap.List` where the schema declares an aggregate, i.e., the `BasicOntology` associates any aggregate with `List`.

5.2 Sending and Receiving Messages

We restrict the description of ontologies to the features that support inter-agent communication. Other models, e.g., DAML+OIL, use description logics to provide richer description that support reasoning about concepts, predicates, actions, etc. In order to send and receive messages, we need *(i)* an ontology to provides the vocabulary and *(ii)* a codec (coder/encoder) to handle the syntax of the content language. These are registered with JADE through the *content manager*. The content manager provides methods for encoding and decoding the content of messages exploiting the registered ontologies and codecs. The following code registers the `People` ontology with the content manager and it also registers a codec called `jade.content.lang.j.JCodec`.

```
getContentManager().registerOntology(PeopleOntology.getInstance());
getContentManager().registerLanguage(new JCodec());
```

The `JCodec` uses Java serialization to encode and decode the content of messages; the `jade.content.lang.leap.LEAPCodec` provides CLDC-compliant encoding and decoding. The choice of the codec is not so relevant because the rest of the API is content-

language independent. The registration of ontologies and codecs is typically provided in the `setup()` method of the agent.

In order to send a message, we have two possibilities: through *concrete objects* or through *abstract descriptors*. The first approach is the easiest to use, but it is limited:

1) we create our content in terms of objects that belongs to the classes that we associated with schemas in the ontology, e.g., `Person` and `FatherOf` classes;
2) we use `fillContent()` in `ContentManager` to fill the content of the message.

The following code inform an agent that *"John lives in London and his only child Bill lives in Paris"*:

```
ACLMessage message = new ACLMessage(ACLMessage.INFORM);

// Set the fields of the ACL message
…

// Create the concrete object representing the content
Man john = new Man();
Man bill = new Man();
john.setName("John");
bill.setName("Bill");

Address johnAddress = new Address();
johnAddress.setCity("London");
john.setAddress(johnAddress);

Address billAddress = new Address();
billAddress.setCity("Paris");
bill.setAddress(billAddress);

FatherOf fatherOf = new FatherOf();
fatherOf.setFather(john);

List children = new ArrayList();
children.add(bill);

fatherOf.setChildren(children);

getContentManager().fillContent(message, fatherOf);
```

Using concrete objects like `john` and `fatherOf` is the easiest approach to filling the content of a message but it is not fully expressive. For examples, consider the following problem: we want to query an agent for the names of John's children. We need to send a query-ref message with the following content: `(iota ?X fatherOf(john, ?X))`, where `?X` is a variable that the receiver agent uses to come to know what we want to know. Such a content is an IRE, i.e., an

expression that identifies an object. The problem is that we cannot set the `children` attribute of a `FatherOf` object to a variable because such an attribute is a `List`. A number of techniques are available to solve this problem exploiting inheritance, but they all require that you implement many classes for describing the ontology. In order to solve this problem using only the classes we already implemented for the ontology, we introduced abstract descriptors. An abstract descriptor is an object that describes an instantiation of a schema, e.g., the following is the abstract descriptor that describes the concept "John":

```
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");
```

An abstract descriptor is created with the name of a class (`MAN` in this example). Then, we can set and get values on the descriptor using the names of the attributes. The structure of the descriptor, i.e., what the available attributes are and what are their values, must be coherent with the schema that the ontology associates with the name of the class (`MAN` in this example). The following is the code for performing the query about the names of John's children:

```
ACLMessage message = new ACLMessage(ACLMessage.QUERY_REF);

// Set the fields of the message
…

// Create the abstract descriptor representing the content
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");

AbsVariable  absX        = new AbsVariable("X")
AbsPredicate absFatherOf = new AbsPredicate(PeopleOntology.FATHER_OF);
absFatherOf.set(PeopleOntology.FATHER,   absJohn);
absFatherOf.set(PeopleOntology.CHILDREN, absX);

AbsIRE absIRE = new AbsIRE(absX, absFatherOf);

getContentManager().fillContent(message, absIRE);
```

The procedure for receiving a message is dual to that of sending a message. We can use both concrete objects and abstract descriptors, and if we try to create a concrete object from a message containing a variable, the content manager throws an `UngroundedException`. The following code handles inform messages:

```
ACLMessage msg = blockingReceive(ACLMessage.INFORM);

// The content of informs do not contain variables
Proposition p = (Proposition)getContentManager().extractContent(msg);
```

```
// Handle the content
if(p instanceof FatherOf) {
  …
}
```

If we want to handle incoming queries, we need to use `extractAbsContent()` to create an abstract descriptor from the message:

```
ACLMessage msg = blockingReceive(ACLMessage.QUERY_REF);

// The content of query-refs do contain variables
AbsIRE absIRE = (AbsIRE)getContentManager().extractAbsContent(msg);

// Handle the content
AbsVariable    absX = absIRE.getVariable();
AbsProposition absP = absIRE.getProposition();
```

3.1 The Agent Platform

The standard model of an agent platform, as defined by FIPA, is represented in the following figure.

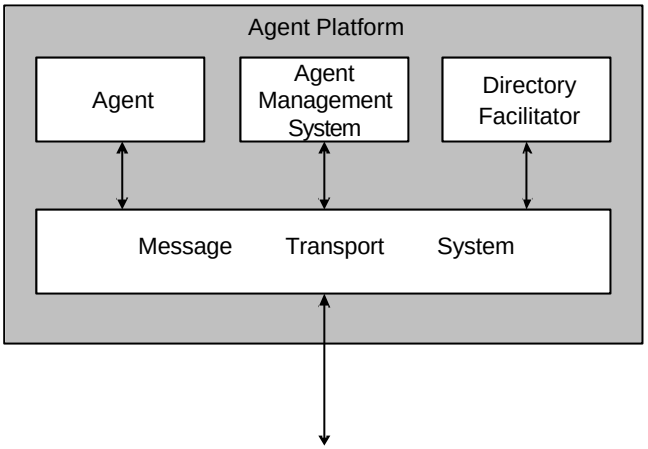


Figure 1 - Reference architecture of a FIPA Agent Platform

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The Message Transport System, also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.

JADE fully complies with this reference architecture and when a JADE platform is launched, the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry, that is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

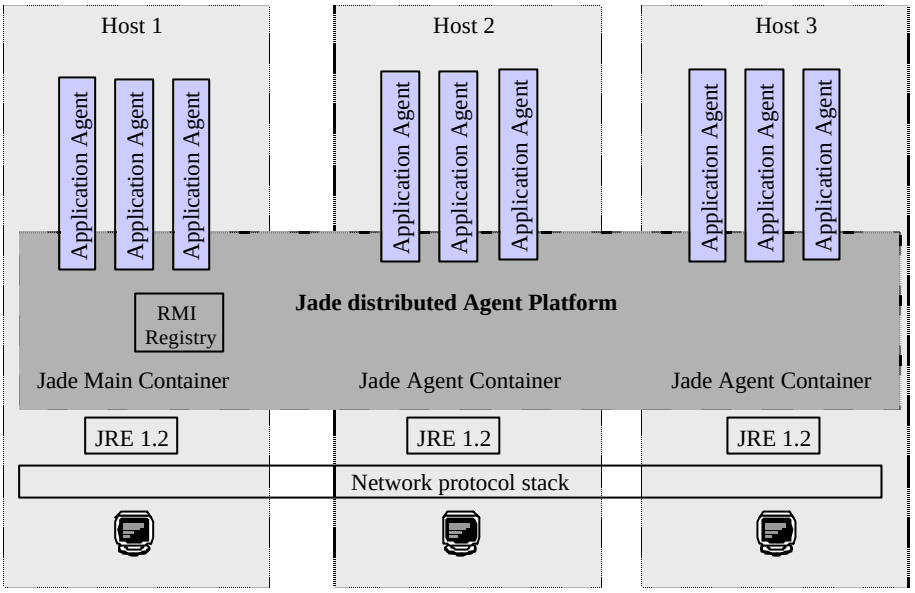


Figure 2 - JADE Agent Platform distributed over several containers

According to the FIPA specifications, DF and AMS agents communicate by using the `FIPA-SL0` content language, the `fipa-agent-management` ontology, and the `fipa-request` interaction protocol. JADE provides compliant implementations for all these components:

- the SL-0 content language is implemented by the class `jade.lang.sl.SL0Codec`. Automatic capability of using this language can be added to any agent by using the method `Agent.registerLanguage(SL0Codec.NAME, new SL0Codec());`
- concepts of the ontology (apart from Agent Identifier, implemented by `jade.core.AID`) are implemented by classes in the `jade.domain.FIPAAgentManagement` package. The `FIPAAgentManagementOntology` class defines the vocabulary with all the constant symbols of the ontology. Automatic capability of using this ontology can be added to any agent by using the fillowing code:
  `Agent.registerOntology(FIPAAgentManagementOntology.NAME, FIPAAgentManagementOntology.instance());`
- finally, the `fipa-request` interaction protocol is implemented as ready-to-use behaviours in the package `jade.proto`.

3.1.1 FIPA-Agent-Management ontology

Every class implementing a concept of the `fipa-agent-management` ontology is a simple collection of attributes, with public methods to read and write them, according to the frame based model that represents FIPA `fipa-agent-management` ontology concepts. The following convention has been used. For each attribute of the class, named `attrName` and of type `attrType`, two cases are possible:

JADE and Ontology

Ontology 101

Excerpt from Ontology 101

JADE provides extensive support for ontologies. The agents df and ams communicate using standard FIPA ontolgies. JADE provides support for user defined ontolgies as well.

Up to JADE 2.3, JADE support for user ontolgies is in the package jade.onto and its subpackages. Starting with version 2.4 a new set of packages, java.content and its subpackages, have made implementing user ontolgies easier. In cps720 we will look at parts of the jade.content packages. The development of ontolgies is a major topic which could easily provide material for a whole course on its own. In cps720 we just look at the tip of this iceberg.

The jade.content Packages

If you look at the API docs for these packages you will find them rather skimpy. The only real documentation is in an appendix to the JADE Programmer's Guide.

Programmer's Guide, Appendix A

This appendix describes a simple ontology example to illustrate the jade.ontology packages. There are three files for this example,

- PeopleOntology.java
- Sender.java
- Receiver.java

All the files for this example are in jadeontology.jar.

Another Ontology Example

A very simple ontology is provided with assignment 3 (Fall 2001). It is packaged in c720a3Ontology.jar. Here is the main file, EconOntology.java.

Excerpts from Ontology 101

From Section 1

Some ontology-design ideas in this guide originated from the literature on object-oriented design(Rumbaugh et al. 1991; Booch et al. 1997). However, **ontology development is different fromdesigning classes and relations in object-oriented programming. Object-oriented programming** centers primarily around methods on classes—a programmer makes design decisions based on the **operational properties** of a class, whereas an **ontology** designer makes these decisions based on the **structural properties** of a class. As a result, a class structure and relations among classes in an ontology are different from the structure for a similar domain in an object-oriented program.

==========>

2 What is in an ontology?

The Artificial-Intelligence literature contains many definitions of an ontology; many of these contradict one another. For the purposes of this guide an ontology is a formal explicit description of concepts in a domain of discourse (**classes (sometimes called concepts)**), properties of each concept describing various features and attributes of the concept **(slots (sometimes called roles** or properties)), and **restrictions on slots (facet**s (sometimes called role restrictions)). **An ontology together with a set of individual instances of classes constitutes a knowledge base**. In reality, there is a fine line where the ontology ends and the knowledge base begins. Classes are the focus of most ontologies. **Classes describe concepts** in the domain. For example, a class of wines represents all wines. Specific wines are instances of this class. The Bordeaux wine in the glass in front of you while you read this document is an instance of the class of Bordeaux wines. A class can have subclasses that represent concepts that are more specific than the superclass. For example, we can divide the class of all wines into red, white, and rosé wines.

Notes/

Beware of confusion with the word class as used in ontology and OO.

In JADE ontologies are, of course, implemented in classes since Java is an OOPS. In the java.content packages the idea of a schema is intorduced which corresponds more closely to the ontology concept of class.

5 APPENDIX A: CONTENT-LANGUAGE INDEPENDENT API
FEDERICO BERGENTI (UNIVERSITY OF PARMA)

Application-specific ontologies describe the elements that agents use to create the content of messages, e.g., application-specific predicates and actions. The package `jade.content` (and its sub-packages) allows to create application-specific ontologies and to use them independently of the adopted content language: the code that implements the ontology and the code that sends and receives messages do not depend on the content language. The following is a description of such a package that uses `src/example/content` as a running example.

5.1 Creating an Application-Specific Ontology

An ontology defines a vocabulary and a set of relationships between the elements of the vocabulary. The relationships can be:

1) structural, e.g., the predicate `fatherOf` is defined over two parameters, a father and a set of children because we want to use it to say `fatherOf(John, (Mary, Lisa))`;
2) semantic, e.g., a concept belonging to the class `Man` also belongs to the class `Person`.

An application-specific ontology is implemented through one object of class `FullOntology` and it is characterized by:

1) one name;
2) one base ontology at most, i.e., an ontology that it extends;
3) a vocabulary;
4) a set of element schemata.

The following code implements the `People` ontology: it defines a constant for each element (concept, action, predicate, etc.) that we want to include in the vocabulary.

```
public class PeopleOntology extends FullOntology {

  // The name of this ontology.
  public static final String ONTOLOGY_NAME = "PEOPLE_ONTOLOGY";

  // Concepts, i.e., objects of the world.
  public static final String PERSON  = "PERSON";
  public static final String MAN     = "MAN";
  public static final String WOMAN   = "WOMAN";
  public static final String ADDRESS = "ADDRESS";

  // Slots of concepts, i.e., attributes of objects.
  public static final String NAME   = "NAME";
  public static final String STREET = "STREET";
  public static final String NUMBER = "NUMBER";
  public static final String CITY   = "CITY";

  // Predicates
  public static final String FATHER_OF = "FATHER_OF";
  public static final String MOTHER_OF = "MOTHER_OF";

```

```
  // Roles in predicates, i.e., names of arguments for predicates
  public static final String FATHER   = "FATHER";
  public static final String MOTHER   = "MOTHER";
  public static final String CHILDREN = "CHILDREN";

  // Actions
  public static final String MARRY = "MARRY";

  // Arguments in actions
  public static final String HUSBAND = "HUSBAND";
  public static final String WIFE    = "WIFE";

  private static PeopleOntology theInstance = new PeopleOntology();

  public static PeopleOntology getInstance() {
    return theInstance;
  }

  public PeopleOntology(Ontology base) {
    super(ONTOLOGY_NAME, ACLOntology.getInstance());

    // Add definitions of schemata here.
    …
  }
}
```

The constructor calls `super()` to assign a name to the ontology and to declare that it extends the `ACLOntology`. `People` ontology, and reasonably all ontologies, extends `jade.content.onto.ACLOntology` because we want to use in our messages the elements of such an ontology, e.g., variables and the `Done` predicate. If you do not need ACL concepts, you can extend `jade.content.onto.BasicOntology` in order to have only basic types, i.e., lists, strings and numbers. `ACLOntology` extends the `BasicOntology`.

The definition of the ontology in the example is not complete, we have to substitute dots with the definition of the element schemata. Element schemata are objects describing the structure of concepts, actions, predicate, etc. that we allow in our messages. In the `People` ontology they describe what a person is, what an address is, what a father is, etc. The following is the element schema for the concept of `Person`. This schema states that a `Person` is characterized by a `name` and an `address`:

```
// Get the element schema for strings from BasicOntology
PrimitiveSchema stringSchema =
  (PrimitiveSchema)getSchema(BasicOntology.STRING);

// Define the concept of Person
ConceptSchema personSchema = new ConceptSchema(PERSON);
personSchema.add(NAME,    stringSchema);
```

```
personSchema.add(ADDRESS, addressSchema, ObjectSchema.OPTIONAL);

// Add the schema to the ontology
add(personSchema);
```

PERSON, NAME and ADDRESS are string defined in the vocabulary and addressSchema has been defined before (not shown). Schemata that describe concepts support inheritance[11]. You can define the concept of Man as a refinement of the concept of Person:

```
ConceptSchema manSchema = new ConceptSchema(MAN);
manSchema.addSuperSchema(personSchema);
```

Element schema describe, in some way, the structure of a class of objects and therefore they can be associated with Java classes. This maps elements of the ontology that comply with a schema with Java objects of that class. The following is a class that might be associated with the Person concept:

```
public class Person extends Concept {
  private String  name    = null;
  private Address address = null;

  public void setName(String name) {
    this.name = name;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

  public String getName() {
    return name;
  }

  public Address getAddress() {
    return address;
  }
}
```

The association between the class and the schema is performed when registering the concept in the ontology using the following statement:

[11] Only concept schemata support inheritance, all other schemata, e.g., predicate schemata, action schemata, etc do not.

```
// Add the schema to the ontology
addElement(personSchema, Person.class);
```

Associating classes with schemata is not mandatory, but helps because it support easier APIs, as shown later. In order to associate a class with a schema, the class must:

1) extend a class in `jade.content`, e.g., `Person` extends `Concept` because we want to associate it with a concept schema;
2) provide public `get`/`set` methods for each attribute (you can use basic types like `int` or `boolean`);
3) provide a constructor with no parameters, i.e., the default contructor.

Defining actions, predicates, etc. is just like defining concepts. Figure 8 shows the classes that corresponds to the elements we can use in our ontologies.

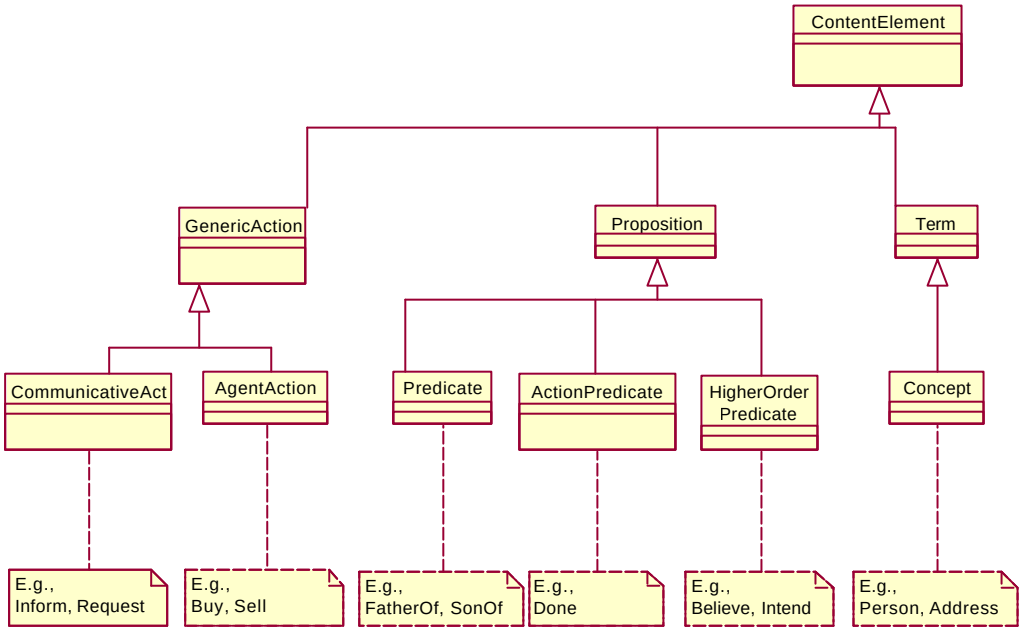


Figure 8 - Classes that correspond to elements of ontologies.

The following is the definition of the predicate `fatherOf`:

```
// Define a schema for the set of children
AggregateSchema childrenSchema = new AggregateSchema(BasicOntology.SET);

// Define the schema for fatherOf predicate
PredicateSchema fatherOfSchema = new PredicateSchema(FATHER_OF);

fatherOfSchema.add(FATHER,   manSchema);
fatherOfSchema.add(CHILDREN, childrenSchema);
```

```
// Add the predicate to the ontology
add(fatherOfSchema, FatherOf.class);
```

First we define a new schema to describe the set of children. Then we define the predicate schema for `fatherOf` by introducing two roles: the `father` and the `children`. Finally we register `fatherOfSchema` with the ontology associating it with the following class:

```
public class FatherOf extends Predicate {
  private List children = null;
  private Man  father   = null;

  public void setChildren(List children) {
    this.children = children;
  }

  public void setFather(Man father) {
    this.father = father;
  }

  public Man getFather() {
    return father;
  }

  public List getChildren() {
    return children;
  }
}
```

Note that we use `jade.util.leap.List` where the schema declares an aggregate, i.e., the `BasicOntology` associates any aggregate with `List`.

5.2 Sending and Receiving Messages

We restrict the description of ontologies to the features that support inter-agent communication. Other models, e.g., DAML+OIL, use description logics to provide richer description that support reasoning about concepts, predicates, actions, etc. In order to send and receive messages, we need *(i)* an ontology to provides the vocabulary and *(ii)* a codec (coder/encoder) to handle the syntax of the content language. These are registered with JADE through the *content manager*. The content manager provides methods for encoding and decoding the content of messages exploiting the registered ontologies and codecs. The following code registers the `People` ontology with the content manager and it also registers a codec called `jade.content.lang.j.JCodec`.

```
getContentManager().registerOntology(PeopleOntology.getInstance());
getContentManager().registerLanguage(new JCodec());
```

The `JCodec` uses Java serialization to encode and decode the content of messages; the `jade.content.lang.leap.LEAPCodec` provides CLDC-compliant encoding and decoding. The choice of the codec is not so relevant because the rest of the API is content-

language independent. The registration of ontologies and codecs is typically provided in the `setup()` method of the agent.

In order to send a message, we have two possibilities: through *concrete objects* or through *abstract descriptors*. The first approach is the easiest to use, but it is limited:

1) we create our content in terms of objects that belongs to the classes that we associated with schemas in the ontology, e.g., `Person` and `FatherOf` classes;
2) we use `fillContent()` in `ContentManager` to fill the content of the message.

The following code inform an agent that *"John lives in London and his only child Bill lives in Paris"*:

```
ACLMessage message = new ACLMessage(ACLMessage.INFORM);

// Set the fields of the ACL message
…

// Create the concrete object representing the content
Man john = new Man();
Man bill = new Man();
john.setName("John");
bill.setName("Bill");

Address johnAddress = new Address();
johnAddress.setCity("London");
john.setAddress(johnAddress);

Address billAddress = new Address();
billAddress.setCity("Paris");
bill.setAddress(billAddress);

FatherOf fatherOf = new FatherOf();
fatherOf.setFather(john);

List children = new ArrayList();
children.add(bill);

fatherOf.setChildren(children);

getContentManager().fillContent(message, fatherOf);
```

Using concrete objects like `john` and `fatherOf` is the easiest approach to filling the content of a message but it is not fully expressive. For examples, consider the following problem: we want to query an agent for the names of John's children. We need to send a query-ref message with the following content: `(iota ?X fatherOf(john, ?X))`, where `?X` is a variable that the receiver agent uses to come to know what we want to know. Such a content is an IRE, i.e., an

expression that identifies an object. The problem is that we cannot set the `children` attribute of a `FatherOf` object to a variable because such an attribute is a `List`. A number of techniques are available to solve this problem exploiting inheritance, but they all require that you implement many classes for describing the ontology. In order to solve this problem using only the classes we already implemented for the ontology, we introduced abstract descriptors. An abstract descriptor is an object that describes an instantiation of a schema, e.g., the following is the abstract descriptor that describes the concept "John":

```
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");
```

An abstract descriptor is created with the name of a class (`MAN` in this example). Then, we can set and get values on the descriptor using the names of the attributes. The structure of the descriptor, i.e., what the available attributes are and what are their values, must be coherent with the schema that the ontology associates with the name of the class (`MAN` in this example). The following is the code for performing the query about the names of John's children:

```
ACLMessage message = new ACLMessage(ACLMessage.QUERY_REF);

// Set the fields of the message
…

// Create the abstract descriptor representing the content
AbsConcept absJohn = new AbsConcept(PeopleOntology.MAN);
absJohn.set(PeopleOntology.NAME, "John");

AbsVariable  absX        = new AbsVariable("X")
AbsPredicate absFatherOf = new AbsPredicate(PeopleOntology.FATHER_OF);
absFatherOf.set(PeopleOntology.FATHER,   absJohn);
absFatherOf.set(PeopleOntology.CHILDREN, absX);

AbsIRE absIRE = new AbsIRE(absX, absFatherOf);

getContentManager().fillContent(message, absIRE);
```

The procedure for receiving a message is dual to that of sending a message. We can use both concrete objects and abstract descriptors, and if we try to create a concrete object from a message containing a variable, the content manager throws an `UngroundedException`. The following code handles inform messages:

```
ACLMessage msg = blockingReceive(ACLMessage.INFORM);

// The content of informs do not contain variables
Proposition p = (Proposition)getContentManager().extractContent(msg);
```

```
// Handle the content
if(p instanceof FatherOf) {
  …
}
```

If we want to handle incoming queries, we need to use `extractAbsContent()` to create an abstract descriptor from the message:

```
ACLMessage msg = blockingReceive(ACLMessage.QUERY_REF);

// The content of query-refs do contain variables
AbsIRE absIRE = (AbsIRE)getContentManager().extractAbsContent(msg);

// Handle the content
AbsVariable    absX = absIRE.getVariable();
AbsProposition absP = absIRE.getProposition();
```

A JADE Ontology

This ontology is part of the JADE exampe using the jade.content packages. It goes with Sender.java and Receiver.java. It is described in appendix A of the JADE Programmer's guide.

PeopleOntology.java (plain text)


```
package examples.content.ontology;

import jade.content.*;
import jade.content.onto.*;
import jade.content.abs.*;
import jade.content.schema.*;
import jade.content.acl.*;
import jade.content.lang.*;

import jade.util.leap.List;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class PeopleOntology extends FullOntology {
    //A symbolic constant, containing the name of this ontology.
    public static final String ONTOLOGY_NAME = "PEOPLE_ONTOLOGY";

    // Concepts
```

```
    public static final String PERSON  = "PERSON";
    public static final String MAN     = "MAN";
    public static final String WOMAN   = "WOMAN";
    public static final String ADDRESS = "ADDRESS";

    // Slots
    public static final String NAME   = "NAME";
    public static final String STREET = "STREET";
    public static final String NUMBER = "NUMBER";
    public static final String CITY   = "CITY";

    // Predicates
    public static final String FATHER_OF = "FATHER_OF";
    public static final String MOTHER_OF = "MOTHER_OF";

    // Roles in predicates
    public static final String FATHER   = "FATHER";
    public static final String MOTHER   = "MOTHER";
    public static final String CHILDREN = "CHILDREN";

    // Actions
    public static final String MARRY = "MARRY";

    // Arguments in actions
    public static final String HUSBAND = "HUSBAND";
    public static final String WIFE    = "WIFE";

    private static PeopleOntology theInstance = new
PeopleOntology(ACLOntology.getInstance());

    public static PeopleOntology getInstance() {
        return theInstance;
    }

    public PeopleOntology(FullOntology base) {
        super(ONTOLOGY_NAME, base);

        try {
            PrimitiveSchema stringSchema  =
(PrimitiveSchema)getSchema(BasicOntology.STRING);
            PrimitiveSchema integerSchema =
(PrimitiveSchema)getSchema(BasicOntology.INTEGER);
```

```
            ConceptSchema addressSchema = new
ConceptSchema(ADDRESS);
            addressSchema.add(STREET, stringSchema,
ObjectSchema.OPTIONAL);
            addressSchema.add(NUMBER, integerSchema,
ObjectSchema.OPTIONAL);
            addressSchema.add(CITY,   stringSchema);

            ConceptSchema personSchema = new
ConceptSchema(PERSON);
            personSchema.add(NAME,    stringSchema);
            personSchema.add(ADDRESS, addressSchema,
ObjectSchema.OPTIONAL);

            ConceptSchema manSchema = new ConceptSchema(MAN);
            manSchema.addSuperSchema(personSchema);

            ConceptSchema womanSchema = new ConceptSchema(WOMAN);
            womanSchema.addSuperSchema(personSchema);

            add(personSchema, Person.class);
            add(manSchema, Man.class);
            add(womanSchema, Woman.class);
            add(addressSchema, Address.class);

            AggregateSchema childrenSchema = new
AggregateSchema(BasicOntology.SET);

            PredicateSchema fatherOfSchema = new
PredicateSchema(FATHER_OF);
            fatherOfSchema.add(FATHER,   manSchema);
            fatherOfSchema.add(CHILDREN, personSchema);

            PredicateSchema motherOfSchema = new
PredicateSchema(MOTHER_OF);
            motherOfSchema.add(CHILDREN, personSchema);

            add(fatherOfSchema, FatherOf.class);
            add(motherOfSchema, MotherOf.class);

            AgentActionSchema marrySchema = new
AgentActionSchema(MARRY);
            marrySchema.add(HUSBAND, manSchema);
```

```
            marrySchema.add(WIFE,    womanSchema);

            add(marrySchema);
        } catch(OntologyException oe) { oe.printStackTrace(); }
    }
}
```

This ontology is supported by a number of Java classes.

- Address.java
- Person.java
- FatherOf.java
- MotherOf.java
- Man.java
- Womna.java
- Marry.java

```
package examples.content.ontology;

import jade.content.*;
import jade.content.onto.*;
import jade.content.abs.*;
import jade.content.schema.*;
import jade.content.acl.*;
import jade.content.lang.*;

import jade.util.leap.List;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class PeopleOntology extends FullOntology {
    //A symbolic constant, containing the name of this ontology.
    public static final String ONTOLOGY_NAME = "PEOPLE_ONTOLOGY";

    // Concepts
    public static final String PERSON  = "PERSON";
    public static final String MAN     = "MAN";
    public static final String WOMAN   = "WOMAN";
    public static final String ADDRESS = "ADDRESS";

    // Slots
    public static final String NAME   = "NAME";
    public static final String STREET = "STREET";
    public static final String NUMBER = "NUMBER";
    public static final String CITY   = "CITY";

    // Predicates
    public static final String FATHER_OF = "FATHER_OF";
    public static final String MOTHER_OF = "MOTHER_OF";

    // Roles in predicates
    public static final String FATHER   = "FATHER";
    public static final String MOTHER   = "MOTHER";
```

```
    public static final String CHILDREN = "CHILDREN";

    // Actions
    public static final String MARRY = "MARRY";

    // Arguments in actions
    public static final String HUSBAND = "HUSBAND";
    public static final String WIFE    = "WIFE";

    private static PeopleOntology theInstance = new
PeopleOntology(ACLOntology.getInstance());

    public static PeopleOntology getInstance() {
        return theInstance;
    }

    public PeopleOntology(FullOntology base) {
        super(ONTOLOGY_NAME, base);

        try {
            PrimitiveSchema stringSchema  =
(PrimitiveSchema)getSchema(BasicOntology.STRING);
            PrimitiveSchema integerSchema =
(PrimitiveSchema)getSchema(BasicOntology.INTEGER);

            ConceptSchema addressSchema = new ConceptSchema(ADDRESS);
            addressSchema.add(STREET, stringSchema,  ObjectSchema.OPTIONAL);
            addressSchema.add(NUMBER, integerSchema, ObjectSchema.OPTIONAL);
            addressSchema.add(CITY,   stringSchema);

            ConceptSchema personSchema = new ConceptSchema(PERSON);
            personSchema.add(NAME,    stringSchema);
            personSchema.add(ADDRESS, addressSchema, ObjectSchema.OPTIONAL);

            ConceptSchema manSchema = new ConceptSchema(MAN);
            manSchema.addSuperSchema(personSchema);

            ConceptSchema womanSchema = new ConceptSchema(WOMAN);
            womanSchema.addSuperSchema(personSchema);

            add(personSchema, Person.class);
            add(manSchema, Man.class);
            add(womanSchema, Woman.class);
            add(addressSchema, Address.class);

            AggregateSchema childrenSchema = new AggregateSchema(BasicOntology.SET);

            PredicateSchema fatherOfSchema = new PredicateSchema(FATHER_OF);
            fatherOfSchema.add(FATHER,   manSchema);
            fatherOfSchema.add(CHILDREN, personSchema);

            PredicateSchema motherOfSchema = new PredicateSchema(MOTHER_OF);
            motherOfSchema.add(CHILDREN, personSchema);

            add(fatherOfSchema, FatherOf.class);
            add(motherOfSchema, MotherOf.class);

            AgentActionSchema marrySchema = new AgentActionSchema(MARRY);
            marrySchema.add(HUSBAND, manSchema);
            marrySchema.add(WIFE,    womanSchema);
```

```
            add(marrySchema);
        } catch(OntologyException oe) { oe.printStackTrace(); }
    }
}
```

```
package examples.content.ontology;

import jade.content.*;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class Address implements Concept {
        private String city   = null;
        private String street = null;
        private int    number = 0;

        public void setCity(String city) {
                this.city = city;
        }

        public void setStreet(String street) {
                this.street = street;
        }

        public void setNumber(int number) {
                this.number = number;
        }

        public String getCity() {
                return city;
        }

        public String getStreet() {
                return street;
        }

        public int getNumber() {
                return number;
        }
}
```

```
package examples.content.ontology;

import jade.content.*;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class Person implements Concept {
        private String  name    = null;
        private Address address = null;

        public void setName(String name) {
                this.name = name;
        }

        public void setAddress(Address address) {
                this.address = address;
        }

        public String getName() {
                return name;
        }

        public Address getAddress() {
                return address;
        }
}
```

```
package examples.content.ontology;

import jade.content.*;

import jade.util.leap.List;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class FatherOf implements Predicate {
        private List children = null;
        private Man  father   = null;

        public void setChildren(List children) {
                this.children = children;
        }

        public void setFather(Man father) {
                this.father = father;
        }

        public Man getFather() {
                return father;
        }

        public List getChildren() {
                return children;
        }
}
```

```
package examples.content.ontology;

import jade.content.*;

import jade.util.leap.List;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class MotherOf implements Predicate {
        private List  children = null;
        private Woman mother   = null;

        public void setChildren(List children) {
                this.children = children;
        }

        public void setMother(Woman mother) {
                this.mother = mother;
        }

        public Woman getMother() {
                return mother;
        }

        public List getChildren() {
                return children;
        }
}
```

```
package examples.content.ontology;


/**
@author Federico Bergenti - Universita` di Parma
*/

public class Man extends Person {}
```

```
package examples.content.ontology;

/**
@author Federico Bergenti - Universita` di Parma
*/

public class Woman extends Person {}
```

JADE Sender Agent

```
/**************************************************************
JADE - Java Agent DEvelopment Framework is a framework to develop
multi-agent systems in compliance with the FIPA specifications.
Copyright (C) 2000 CSELT S.p.A.

GNU Lesser General Public License

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation,
version 2.1 of the License.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the
Free Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA  02111-1307, USA.
**************************************************************/
```

```
package examples.content;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

import jade.util.leap.List;
import jade.util.leap.ArrayList;

import jade.content.*;
import jade.content.abs.*;
import jade.content.onto.*;
import jade.content.lang.*;
import jade.content.lang.leap.*;

import examples.content.ontology.*;

public class Sender extends Agent {
    // We handle contents
    private ContentManager manager  =
(ContentManager)getContentManager();
    // This agent speaks a language called "LEAP"
```

```
    private Codec          codec    = new LEAPCodec();
    // This agent complies with the People ontology
    private FullOntology   ontology =
PeopleOntology.getInstance();

    class SenderBehaviour extends SimpleBehaviour {
    private boolean finished = false;

    public SenderBehaviour(Agent a) { super(a); }

    public boolean done() { return finished; }

    public void action() {
        try {
        // Preparing the first message
        System.out.println( "[" + getLocalName() + "] Creating
inform message with content fatherOf(man :name John :address
London, [man :name Bill :address Paris])");

        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        AID receiver = new AID("receiver", false);

        msg.setSender(getAID());
        msg.addReceiver(receiver);
        msg.setLanguage(codec.getName());
        msg.setOntology(ontology.getName());

        // The message informs that:
        // fatherOf(man :name "John" :address "London", [man :name
"Bill" :address "Paris"])

        Man john = new Man();
        Man bill = new Man();
        john.setName("John");
        bill.setName("Bill");

        Address johnAddress = new Address();
        johnAddress.setCity("London");
        john.setAddress(johnAddress);

        Address billAddress = new Address();
        billAddress.setCity("Paris");
        bill.setAddress(billAddress);
```

```
        FatherOf fatherOf = new FatherOf();
        fatherOf.setFather(john);

        List children = new ArrayList();
        children.add(bill);

        fatherOf.setChildren(children);

        // Fill the content of the message
        manager.fillContent(msg, fatherOf);

        // Send the message
        System.out.println( "[" + getLocalName() + "] Sending the
message...");
        send(msg);

        // Now ask the proposition back.
        // Use a query-ref with the following content:
        // iota ?x fatherOf(?x, [man :name "Bill" :address
"Paris"])
        System.out.println( "[" + getLocalName() + "] Creating
query-ref message with content iota ?x fatherOf(?x, [man :name
Bill :address Paris])");
        msg.setPerformative(ACLMessage.QUERY_REF);

        // Create an abstract descriptor from scratch
        AbsConcept absBill = new AbsConcept(PeopleOntology.MAN);
        absBill.set(PeopleOntology.NAME, "Bill");

        // Create an abstract descriptor from a concrete object
        AbsConcept absBillAddress =
(AbsConcept)ontology.fromObject(billAddress);
        absBill.set(PeopleOntology.ADDRESS, absBillAddress);

        AbsAggregate absChildren = new
AbsAggregate(BasicOntology.SET);
        absChildren.add(absBill);

        AbsVariable absX = new AbsVariable("x",
PeopleOntology.MAN);

        AbsPredicate absFatherOf = new
AbsPredicate(PeopleOntology.FATHER_OF);
        absFatherOf.set(PeopleOntology.FATHER, absX);
```

```
        absFatherOf.set(PeopleOntology.CHILDREN, absChildren);

        AbsIRE absIRE = new AbsIRE();
        absIRE.setVariable(absX);
        absIRE.setKind(ACLOntology.IOTA);
        absIRE.setProposition(absFatherOf);

            // Fill the content of the message
        manager.fillContent(msg, absIRE);

        // Send the message
        System.out.println( "[" + getLocalName() + "] Sending the
message...");
        send(msg);
        } catch(Exception e) { e.printStackTrace(); }

        finished = true;
    }
    }

    protected void setup() {
    manager.registerLanguage(codec);
    manager.registerOntology(ontology);

    addBehaviour(new SenderBehaviour(this));
    }
}
```

JADE Receiver Agent

```
/**************************************************************
JADE - Java Agent DEvelopment Framework is a framework to develop
multi-agent systems in compliance with the FIPA specifications.
Copyright (C) 2000 CSELT S.p.A.

GNU Lesser General Public License

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation,
version 2.1 of the License.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the
Free Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA  02111-1307, USA.
**************************************************************/

package examples.content;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

import jade.content.*;
import jade.content.abs.*;
import jade.content.onto.*;
import jade.content.lang.*;
import jade.content.lang.leap.*;


import examples.content.ontology.*;

public class Receiver extends Agent {
    private ContentManager manager     =
(ContentManager)getContentManager();
    private Codec          codec       = new LEAPCodec();
    private FullOntology   ontology    =
```

```
PeopleOntology.getInstance();
    private FatherOf       proposition = null;

    class ReceiverBehaviour extends SimpleBehaviour {
    private boolean finished = false;

    public ReceiverBehaviour(Agent a) { super(a); }

    public boolean done() { return finished; }

    public void action() {
        for(int c = 0; c < 2; c++) {
        try {
            System.out.println( "[" + getLocalName() + "] Waiting
for a message...");

            ACLMessage msg = blockingReceive();

            if (msg!= null) {
            switch(msg.getPerformative()) {
            case ACLMessage.INFORM:
                ContentElement p = manager.extractContent(msg);
                if(p instanceof FatherOf) {
                    proposition = (FatherOf)p;
                    System.out.println("[" + getLocalName() + "]
Receiver inform message: information stored.");
                   System.out.println("Father name " +
proposition.getFather().getName());
                    break;
                }
            case ACLMessage.QUERY_REF:
                AbsContentElement abs =
manager.extractAbsContent(msg);
                if(abs instanceof AbsIRE) {
                    AbsIRE ire = (AbsIRE)abs;

                    ACLMessage reply = new
ACLMessage(ACLMessage.INFORM);
                    AID sender = new AID("sender", false);

                    msg.setSender(getAID());
                    msg.addReceiver(sender);
                    msg.setLanguage(codec.getName());
```

```
                    msg.setOntology(ontology.getName());

                    AbsConcept absFather =
(AbsConcept)ontology.fromObject(proposition.getFather());

                    AbsEquals absEquals = new AbsEquals();
                    absEquals.setIRE(ire);
                    absEquals.setConcept(absFather);

                    manager.fillContent(msg, absEquals);

                    send(msg);

                    System.out.println("[" + getLocalName() + "]
Received query-ref message: reply sent:");
                    absEquals.dump();
                    break;
                }
            default:
                System.out.println("[" + getLocalName() + "]
Malformed message.");
            }
            }
        } catch(Exception e) { e.printStackTrace(); }
        }
        finished = true;
    }
    }

    protected void setup() {
        manager.registerLanguage(codec);
        manager.registerOntology(ontology);

        addBehaviour(new ReceiverBehaviour(this));
    }
}
```

A Simple JADE Ontology for Economics

```
package cps720.assignment3.ontology;

import jade.content.*;
import jade.content.onto.*;
import jade.content.abs.*;
import jade.content.schema.*;
import jade.content.acl.*;
import jade.content.lang.*;

/**
 * An ontology for the supply-demand simulation. (See
Producer.java and Consumer.java.)
 *
 * These ontologies are quite confusing. You have to link the
"schemas" to the classes using
 * the static string constants. The different types of schemas are
related to the categories of
 * speech acts such as predicateas and actions.
 * (see more comments below>)
 *
 * DG. October, 2001
 */
public class EconOntology extends FullOntology {

    // A name for the ontology -- passed to the super class
constructor
    public static final String ONTOLOGY_NAME = "ECON_ONTOLOGY";

    // concepts (classes)

    public static final String  PRODUCT = "PRODUCT";

    // roles (slots)

   public static final String NAME = "NAME";
    public static final String PRICE = "PRICE";
    public static final String QUANTITY = "QUANTITY";
    public static final String UNIT_COST = "UNIT_COST";
    public static final String VALUE = "VALUE";


    // predicates

    public static final String PRICE_OF =  "PRICE_OF";
    public static final String QUANTITY_OF = "QUANTITY_OF";
```

```
   // actions

    public static final String BUY = "BUY";

    // argumements for actions

    public static final String PURCHASE = "PURCHASE";

      //
=========================================================================

    // Some JADE setup methods

    private static EconOntology thisInstance = new
EconOntology(ACLOntology.getInstance());

    public static EconOntology getInstance() {
        return thisInstance;
    }

    public EconOntology(FullOntology base) {

        super(ONTOLOGY_NAME, base);

        try {
            // include two data types
            PrimitiveSchema stringSchema  =
(PrimitiveSchema)getSchema(BasicOntology.STRING);
            PrimitiveSchema floatSchema =
(PrimitiveSchema)getSchema(BasicOntology.FLOAT);

            /*
             * Concepts are objects of the ontology (abstract or
concrete). They are the nouns.
             *
             * The add() method adds slots, sometimes called
facets.
             */
            ConceptSchema productSchema = new
ConceptSchema(PRODUCT);
               productSchema.add(NAME, stringSchema);
               productSchema.add(PRICE, floatSchema);
               productSchema.add(UNIT_COST, floatSchema,
ObjectSchema.OPTIONAL);
               productSchema.add(QUANTITY, floatSchema,
ObjectSchema.OPTIONAL); //eg tons of wheat

            /**
             * Each concept in the ontology is associated with a
```

```
Java class.
             */
            add(productSchema, Product.class);

            /*
             * Predicates have truth values and express relations
among the concepts.
             * The string constants (e.g., PRICE_OF) names the
predicate for the JADE sysetm.
             */
            PredicateSchema priceOfSchema = new
PredicateSchema(PRICE_OF);
            PredicateSchema quantityOfSchema = new
PredicateSchema(QUANTITY_OF);

            /**
             * Now add the concepts involved in the predicate. You
need the string name constant for
             * each concept, and, a corresponding schema.
             */
            priceOfSchema.add(PRODUCT, productSchema);
            priceOfSchema.add(PRICE, floatSchema);

            quantityOfSchema.add(PRODUCT, productSchema);
            quantityOfSchema.add(QUANTITY,floatSchema);

            /*
             * And associate a Java class.
             */
            add(priceOfSchema, PriceOf.class);
            add(quantityOfSchema, QuantityOf.class);

            /*
            * You may also have actions. These are handled in the
same way.
             */
            AgentActionSchema buySchema = new
AgentActionSchema(BUY);
            buySchema.add(PRODUCT, productSchema);

            add(buySchema, Buy.class);

        } catch (OntologyException oe) {
            oe.printStackTrace();
        }

    }

}
```

Product.java

A support file for the EconOntology.

```
package cps720.assignment3.ontology;

import jade.content.*;

/**
 * Part of the EconOntolgy
 * setters and getters.
 * DG. October, 2001
 */
public class  Product implements Concept {

    private String name = null;
    private float price = 0.0f;
    private float unitCost = 0.0f;
    private float quantity = 0.0f;

    public String getName() {
        return name;
    }
    public float  getPrice() {
        return price;
    }
    public float getQuantity() {
        return quantity;
    }
    public float getUnitCost() {
        return unitCost;
    }

    public void setName(String n) {
        name = n;
    }
    public void setPrice(float p) {
        price = p;
    }
    public void setQuantity(float q) {
        quantity = q;
    }
```

```
    public void setUnitCost(float c) {
        unitCost = c;
    }
}
```

Some JADE Examples

The following pages discuss some JADE examples. The first is a very simple example. The second is much more elaborate. The third uses a custom ontolgy

Ping Agent

Party Agents

Ontology Example

JADE Ping Agent

This is a very simple agent. After loading it into a container, run the Dummy Agent and use it to send the message "ping" (do not enter the quotes) with performative QUERY-REF or QUERY-IF. Also try it with some other performative to see what happens.

PingAgent.java

(Important classes in green (teal). Methods called by JADE in red.)


```
package examples.PingAgent;

import java.util.Date;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.OutputStreamWriter;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;


/**
This agent implements a simple Ping Agent.
First of all the agent registers itself with the DF of the platform and
then waits for ACLMessages.
If  a QUERY_REF or QUER_IF message arrives that contains the string "ping" within the content
then it replies with an INFORM message whose content will be the string "(pong)".
If it receives a NOT_UNDERSTOOD message no reply is sent.
For any other message received it replies with a NOT_UNDERSTOOD message.
```

The exchanged message are written in a log file whose name is the local name of the agent.

@author Tiziana Trucco - CSELT S.p.A.
@version $Date: 2001/02/09 16:17:09 $ $Revision: 1.2 $
*/

```
public class PingAgent extends Agent {

   PrintWriter logFile;

  class WaitPingAndReplyBehaviour extends CyclicBehaviour {

   public WaitPingAndReplyBehaviour(Agent a) {
      super(a);
    }

    public void action() {

      ACLMessage  msg = blockingReceive();


      if(msg != null){
        if(msg.getPerformative() == ACLMessage.NOT_UNDERSTOOD)
        {

            log("Received the following message: "+
msg.toString());
            log("No reply message sent.");
        }
        else{
            log("Received the following message: "+
msg.toString());
           ACLMessage reply = msg.createReply();

           if((msg.getPerformative()==
ACLMessage.QUERY_REF)||(msg.getPerformative()==
ACLMessage.QUERY_IF))
           {
             String content = msg.getContent();
             if ((content != null) && (content.indexOf("ping") !=
-1))
            {{
               reply.setPerformative(ACLMessage.INFORM);
               reply.setContent("(pong)");
             }
            eelse
```

```
                {
                  reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                  reply.setContent("( UnexpectedContent (expected
ping))");
                }
              }
              else
              {
                  reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                  reply.setContent("( (Unexpected-act
"+ACLMessage.getPerformative(msg.getPerformative())+") ( expected
(query-ref :content ping)))");
              }
              log("Replied with the following message: "+ reply.toString());

              send(reply);
          }

        }else{
          //System.out.println("No message received");
        }
     }
    } //Endinner  class WaitPingAndReplyBehaviour

  protected void setup() {

        /** Registration with the DF */

        DFAgentDescription dfd = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();

        sd.setType("PingAgent");
        sd.setName(getName());
        sd.setOwnership("ExampleReceiversOfJADE");
        sd.addOntologies("PingAgent");
        dfd.setName(getAID());
        dfd.addServices(sd);
        try {
          DFService.register(this,dfd);
        } catch (FIPAException e) {
          System.err.println(getLocalName()+" registration with DF
unsucceeded. Reason: "+e.getMessage());
          doDelete();
        }
        try{
```

```
            logFile = new PrintWriter(new
FileWriter(getLocalName()+".log",true));
            log("Agent: " + getName() + " born");

            WaitPingAndReplyBehaviour PingBehaviour = new
WaitPingAndReplyBehaviour(this);

            addBehaviour(PingBehaviour);

        }catch(IOException e){
            System.out.println("WARNING: The agent needs the "+
getLocalName()+".log file.");
            e.printStackTrace();
        }
  }

    public synchronized void log(String str) {
        logFile.println((new Date()).toString()+ " - " + str);
        logFile.flush();
    }
```

}//end class PingAgent

Running PingAgent

You can start the PingAgent with java jade.Boot -gui ping:examples.PingAgent.PingAgent. (Assuming you have your classpaths correct.) You can then use the DummyAgent to send it the "ping" message with performative QUERY_REF or QUERY_IF.

Or you could have another agent send it the "ping" message. For an example, see the file SendPing2.java.

```
package examples.PingAgent;

import java.util.Date;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.OutputStreamWriter;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

/**
This agent implements a simple Ping Agent.
First of all the agent registers itself with the DF of the platform and
then waits for ACLMessages.
If  a QUERY_REF or QUER_IF message arrives that contains the string "ping" within the
content
then it replies with an INFORM message whose content will be the string "(pong)".
If it receives a NOT_UNDERSTOOD message no reply is sent.
For any other message received it replies with a NOT_UNDERSTOOD message.
The exchanged message are written in a log file whose name is the local name of the
agent.

@author Tiziana Trucco - CSELT S.p.A.
@version  $Date: 2001/02/09 16:17:09 $ $Revision: 1.2 $
*/


public class PingAgent extends Agent {

    PrintWriter logFile;

  class WaitPingAndReplyBehaviour extends CyclicBehaviour {
```

```
  public WaitPingAndReplyBehaviour(Agent a) {
     super(a);
   }

   public void action() {

       ACLMessage  msg = blockingReceive();

     if(msg != null){
       if(msg.getPerformative() == ACLMessage.NOT_UNDERSTOOD)
       {
           log("Received the following message: "+ msg.toString());
           log("No reply message sent.");
       }
       else{
           log("Received the following message: "+ msg.toString());
           ACLMessage reply = msg.createReply();

           if((msg.getPerformative()== ACLMessage.QUERY_REF)||(msg.getPerformative()==
ACLMessage.QUERY_IF))
           {
             String content = msg.getContent();
           if ((content != null) && (content.indexOf("ping") != -1))
                      {
                          reply.setPerformative(ACLMessage.INFORM);
                          reply.setContent("(pong)");
                      }
                  else
                  {
                      reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
                      reply.setContent("( UnexpectedContent (expected ping))");
                  }

           }
           else
           {
               reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
               reply.setContent("( (Unexpected-act
"+ACLMessage.getPerformative(msg.getPerformative())+") ( expected (query-ref :content
ping)))");

           }

           log("Replied with the following message: "+ reply.toString());
         send(reply);
       }
     }else{
       //System.out.println("No message received");
     }
   }

  } //End class WaitPingAndReplyBehaviour


 protected void setup() {

       /** Registration with the DF */
     DFAgentDescription dfd = new DFAgentDescription();
       ServiceDescription sd = new ServiceDescription();
       sd.setType("PingAgent");
```

```
        sd.setName(getName());
        sd.setOwnership("ExampleReceiversOfJADE");
        sd.addOntologies("PingAgent");
        dfd.setName(getAID());
        dfd.addServices(sd);
        try {
        DFService.register(this,dfd);
        } catch (FIPAException e) {
        System.err.println(getLocalName()+" registration with DF unsucceeded. Reason:
"+e.getMessage());
        doDelete();
        }

        try{
            logFile = new PrintWriter(new FileWriter(getLocalName()+".log",true));
            log("Agent: " + getName() + " born");
            WaitPingAndReplyBehaviour PingBehaviour = new
WaitPingAndReplyBehaviour(this);
            addBehaviour(PingBehaviour);
        }catch(IOException e){
            System.out.println("WARNING: The agent needs the "+ getLocalName()+".log
file.");
            e.printStackTrace();
        }
  }

    public synchronized void log(String str) {

        logFile.println((new Date()).toString()+ " - " + str);
    logFile.flush();
    }

}//end class PingAgent
```

```
package examples.PingAgent;

import jade.core.*;
import jade.core.behaviours.SimpleBehaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class SendPing2 extends Agent {

    protected  void setup() {

        DFAgentDescription dfd = new DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("SendPing"); // something must be here; otherwise won't register
        sd.setName(getName());
        //sd.addOntologies("PingAgent"); // simple strings ok
        dfd.setName(getAID());
        dfd.addServices(sd);

        try {
            DFService.register(this, dfd);
        } catch (FIPAException e) {
            System.err.println(getLocalName() + " registration with DF failed. Reason: "
+  e.getMessage());
            doDelete();
        }
         addBehaviour(new SimpleBehaviour() {

                private boolean finished = false;

                public void action() {
                    System.out.println("Enter the message 'ping'.");
                    String line = null;
                    try {
                        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                        line = br.readLine();
                    } catch (IOException ioe) {
                        ioe.printStackTrace();
                    }
                     ACLMessage msg = new ACLMessage(ACLMessage.QUERY_REF);
                     msg.setContent(line);
                     msg.setSender(getAID());
                     AID pingAgent = new AID("ping", false);
                     msg.addReceiver(pingAgent);
                     send(msg);

                     msg = blockingReceive();
                     if(msg != null) {
                         if(msg.getPerformative() == ACLMessage.INFORM ) {
                            System.out.println("[" + msg.getSender().getName()+ "] says "
+ msg.getContent());
                        } else if(msg.getPerformative() == ACLMessage.NOT_UNDERSTOOD) {
```

```
                            System.out.println("[" + msg.getSender().getName()+ "] says "
+ msg.getContent());
                             System.out.println("Not understood ?? This is the end!!");
                             finished = true;
                        } else {
                             System.out.println("A mysterious message");
                        }
                    }
                } // end action()

                public boolean done() {
                    return finished;
                }
            });
    }



}
```

The JADE Party

(This agent is not part of the JADE 2.4 distribution.)

The HostAgent creates n party guests who pass a rumour among themselves (see below). The program was originally designed as "stress test" of how well JADE runs on various systems. Do not make n too large!

The program illustrates, among other things, how to attach a user GUI to a JADE agent with the help of OneShotBehavioiurs.

Plain source code

(If you run this make sure you put them in a directory com\hp\hpl\jade_test.)

- HostAgent.java
- HostUIFrame.java
- GuestAgent.java

See Also

- HostUIFrame.
- GuestAgent

HostAgent

```
/*****************************************************************************
 * Source code information
 * -----------------------
 * Original author    Ian Dickinson, HP Labs Bristol
 * Author email       Ian_Dickinson@hp.com
 * Package
 * Created            1 Oct 2001
 * Filename           $RCSfile:  $
 * Revision           $Revision:  $
 * Release status     Experimental. $State: $
 *
** Last modified on   $Date:  $
 *               by   $Author: $
 *
 * Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
 *****************************************************************************/
```

```
package com.hp.hpl.jade_test;

import jade.core.AID;
import jade.core.Agent;
import jade.core.ProfileImpl;
import jade.core.Profile;
import jade.wrapper.AgentContainer;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.core.behaviours.CyclicBehaviour;
```

```
import jade.core.behaviours.OneShotBehaviour;

import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

import javax.swing.*;
import java.util.*;
import java.text.NumberFormat;
/**
** <p>
 * Agent representing the host for a party, to which a user-controlled number of
guests is invited.  The sequence is
 * as follows: the user selects a number guests to attend the party from 0 to 1000,
using the
 * slider on the UI.  When the party starts, the host creates N guest agents, each of
which registers
 * with the DF, and sends the host a message to say that they have arrived.  When all
the guests
 * have arrived, the party starts.  The host selects one guest at random, and tells
them a rumour.
 * The host then selects two other guests at random, and introduces them to each
other.  The party
 * then proceeds as follows: each guest that is introduced to someone asks the host
to introduce them
 * to another guest (at random).  If a guest has someone introduce themselves, and
the guest knows
 * the rumour, they tell the other guest.  When a guest hears the rumour for the
first time, they
 * notify the host.  When all the guests have heard the rumour, the party ends and
the guests leave.
 * </p>
 * <p>
 * Note: to start the host agent, it must be named 'host'.  Thus:
 * <code><pre>
 *     java jade.Boot -gui host:com.hp.hpl.jade_test.HostAgent()
 * </pre></code>
 * </p>
 *
 * @author Ian Dickinson, HP Labs (<a href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */

public class HostAgent
    extends Agent
{

    public final static String HELLO = "HELLO";
    public final static String ANSWER = "ANSWER";
    public final static String THANKS = "THANKS";
```

```
    public final static String GOODBYE = "GOODBYE";
    public final static String INTRODUCE = "INTRODUCE";
    public final static String RUMOUR = "RUMOUR";

    // Instance variables
    ////////////////////////////////

    protected JFrame m_frame = null;
    protected Vector m_guestList = new Vector();    // invitees
    protected int m_guestCount = 0;                 // arrivals
    protected int m_rumourCount = 0;

    protected int m_introductionCount = 0;
    protected boolean m_partyOver = false;
    protected NumberFormat m_avgFormat =
NumberFormat.getInstance();
    protected long m_startTime = 0L;

    // Constructors
    ////////////////////////////////
    /**
     * Construct the host agent.  Some tweaking of the UI
parameters.
     */
    public HostAgent() {
        m_avgFormat.setMaximumFractionDigits( 2 );
        m_avgFormat.setMinimumFractionDigits( 2 );
    }

    // External signature methods
    ////////////////////////////////
    /**
     * Setup the agent.  Registers with the DF, and adds a
behaviour to
     * process ncoming messages.
     */
    protected void setup() {
        try {
            System.out.println( getLocalName() + " setting up");
            // create the agent descrption of itself

            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName( getAID() );
            DFService.register( this, dfd );
```

```
            // add the GUI
            setupUI();

            // add a Behaviour to handle messages from guests

            addBehaviour( new CyclicBehaviour( this ) {
                              public void action() {
                                  ACLMessage msg = receive();
                                  if (msg != null) {
                                      if (HELLO.equals(
msg.getContent() )) {
                                          // a guest has arrived
                                          m_guestCount++;
                                          setPartyState( "Inviting
guests (" + m_guestCount + " have arrived)" );
                                          if (m_guestCount ==
m_guestList.size()) {
                                              System.out.println(
"All guests have arrived, starting conversation" );
                                              // all guests have
arrived

                                              beginConversation();
                                          }
                                      }
                                      else if (RUMOUR.equals(
msg.getContent() )) {

                                          // count the agents who
have heard the rumour
                                          incrementRumourCount();
                                      }
                                      else if (msg.getPerformative()
== ACLMessage.REQUEST  &&  INTRODUCE.equals( msg.getContent() )) {
                                          // an agent has requested
an introduction
                                          doIntroduction(
msg.getSender() );
                                      }
                                  }
                                  else {
                                      // if no message is arrived,
block the behaviour
                                      block();
```

```
                                    }
                                }
                            } );
        }
        catch (Exception e) {
            System.out.println( "Saw exception in HostAgent: " + e
);
            e.printStackTrace();
        }
    }

 s   /**
     * Setup the UI, which means creating and showing the main
frame.
     */
    private void setupUI() {

        m_frame = new HostUIFrame( this );
        m_frame.setSize( 400, 200 );
        m_frame.setLocation( 400, 400 );
        m_frame.setVisible( true );
        m_frame.validate();
    }

    /**
     * Invite a number of guests, as determined by the given
parameter.  Clears old
     * state variables, then creates N guest agents.  A list of
the agents is maintained,
     * so that the host can tell them all to leave at the end of
the party.
    *
     * DG: This and several other methods are invokded by a
OneShotBehaviour of this
     *agents defined in HostUIFrame.java.
     *
     * @param nGuests The number of guest agents to invite.
     */
    protected void inviteGuests( int nGuests ) {

        // remove any old state
        m_guestList.clear();
        m_guestCount = 0;
        m_rumourCount = 0;
        m_introductionCount = 0;
```

```
        m_partyOver = false;
        ((HostUIFrame) m_frame).lbl_numIntroductions.setText( "0"
);
        ((HostUIFrame) m_frame).prog_rumourCount.setValue( 0 );
        ((HostUIFrame) m_frame).lbl_rumourAvg.setText( "0.0" );

        // notice the start time
        m_startTime = System.currentTimeMillis();

        // try first to ensurat the DF has finished deregistering
the old guests
        if (checkDF()) {
            setPartyState( "Inviting guests" );
            try {
                for (int i = 0;  i < nGuests;  i++) {
                    // create a new agent
                    Agent guest = new GuestAgent();
                   // DG: each new agent gets a unique local name.
                    guest.doStart( "guest_" + i );

                    // keep the guest's ID on a local list
                    m_guestList.add( guest.getAID() );

                    checkDF(); // FIXME: Tiziana Trucco
                }
            }
            catch (Exception e) {
                System.err.println( "Exception while adding
guests: " + e );
                e.printStackTrace();
            }
        }
    }

    /**
     * End the party: set the state variables, and tell all the
guests to leave.
     */

    protected void endParty() {

        setPartyState( "Party over" );
        m_partyOver = true;

        // log the duration of the run
        System.out.println( "Simulation run complete. NGuests = "
```

```
+ m_guestCount + ", time taken = " +
                            m_avgFormat.format( ((double)
System.currentTimeMillis() - m_startTime) / 1000.0 ) + "s" );

        // send a message to all guests to tell them to leave
        for (Iterator i = m_guestList.iterator();  i.hasNext();  )
{

            ACLMessage msg = new ACLMessage( ACLMessage.INFORM );
            msg.setContent( GOODBYE );
            msg.addReceiver( (AID) i.next() );

            send(msg);
        }

        m_guestList.clear();
    }

    /**
     * Shut down the host agent, including removing the UI and
deregistering
     * from the DF.
     */
    protected void terminateHost() {
        try {
            if (!m_guestList.isEmpty()) {
                endParty();
            }
            DFService.deregister( this );
            m_frame.dispose();
            doDelete();
        }
        catch (Exception e) {
            System.err.println( "Saw FIPAException while
terminating: " + e );
            e.printStackTrace();
        }
    }
    /**
     * Start the conversation in the party.  Tell a random guest a
rumour, and
     * select two random guests and introduce them to each other.
     */
    protected void beginConversation() {
```

```
        // start a rumour
        ACLMessage rumour = new ACLMessage( ACLMessage.INFORM );
        rumour.setContent( RUMOUR );
        rumour.addReceiver( randomGuest( null ) )

        send( rumour );

        // introduce two agents to each other
        doIntroduction( randomGuest( null ) );

        setPartyState( "Swinging" );
    }

    /**
     * Introduce guest0 to a random other guest.  Also updates the
introduction
     * count on the UI, and the avg no of introductions per
rumour.
     */
    protected void doIntroduction( AID guest0 ) {
        if (!m_partyOver) {
            AID guest1 = randomGuest( guest0 );

            // introduce two guests to each other
            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( INTRODUCE + " " + guest0 );
            m.addReceiver( guest1 );

            send( m );

            // update the count of introductions on the UI
            m_introductionCount++;
            SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).lbl_numIntroductions.setText( Integer.toString(
m_introductionCount ));
                                        }
                                    } );
            updateRumourAvg();
        }
    }

    /**
     * Increment the number of guests that have heard the rumour,
and update the UI.
```

```
     * If all guests have heard the rumour, end the party.
     */
    protected void incrementRumourCount() {
        m_rumourCount++;
        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).prog_rumourCount.setValue( Math.round( 100 *
m_rumourCount / m_guestCount ) );
                                        }
                                    } );
        updateRumourAvg();

        // when all the guests have heard the rumour, the party
ends
        if (m_rumourCount == m_guestCount) {
            // simulate the user clicking stop when the guests
have all heard the rumour
            try {
                SwingUtilities.invokeAndWait( new Runnable() {
                                                  public void run()
{
                                                      ((HostUIFrame)
m_frame).btn_stop_actionPerformed( null );
                                                  }
                                              } );
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * Update the state of the party in the UI
     */
    protected void setPartyState( final String state ) {
        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).lbl_partyState.setText( state );
                                        }
                                    } );
    }
```

```
    /**
     * Update the average number of introductions per rumour
spread
     * in the UI.
     */
    protected void updateRumourAvg() {

        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).lbl_rumourAvg.setText( m_avgFormat.format( ((double)
m_introductionCount) / m_rumourCount ) );
                                        }
                                    } );
    }

    /**
     * Pick a guest at random who is not the given guest.
     *
     * @param aGuest A guest at the party or null
     * @return A random guest who is not aGuest.
     */
    protected AID randomGuest( AID aGuest ) {

        AID g = null;

        do {
            int i = (int) Math.round( Math.random() *
(m_guestList.size() - 1) );
            g = (AID) m_guestList.get( i );
        } while (g == aGuest);

        return g;
    }
    /**
     * Answer true if the DF has cleared all guests.  Note: this
approach does
     * not work at the moment.
     */
    protected boolean checkDF() {
        try {
            ServiceDescription sd = new ServiceDescription();
            sd.setType( "PartyGuest" );
            sd.setName( "GuestServiceDescription" );
            DFAgentDescription dfd = new DFAgentDescription();
```

```
            dfd.addServices( sd );
            DFAgentDescription[] agents = DFService.search( this,
dfd );

            // if not clear, warn user
            if (agents.length > 0) {
                JOptionPane.showMessageDialog( m_frame, "DF has
not finished removing "+ agents.length +" old guest agents, please
be patient", "Warning", JOptionPane.WARNING_MESSAGE );
            }
            else {
                return true;
            }
        }
        catch (Exception e) {
            System.err.println( "Exception: " + e );
            e.printStackTrace();
        }
        return false;
    }

}
```

- HostUIFrame.
- GuestAgent

```
/*****************************************************************************
 * Source code information
 * -----------------------
 * Original author    Ian Dickinson, HP Labs Bristol
 * Author email       Ian_Dickinson@hp.com
 * Package
 * Created            1 Oct 2001
 * Filename           $RCSfile:  $
 * Revision           $Revision:  $
 * Release status     Experimental. $State: $
 *
 * Last modified on   $Date:  $
 *               by   $Author: $
 *
 * Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
 *****************************************************************************/

// Package
///////////////
package com.hp.hpl.jade_test;



// Imports
///////////////
import jade.core.AID;
import jade.core.Agent;
import jade.core.ProfileImpl;
import jade.core.Profile;

import jade.wrapper.AgentContainer;

import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

import jade.core.behaviours.CyclicBehaviour;
import jade.core.behaviours.OneShotBehaviour;

import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.DFService;
import jade.domain.FIPAException;

import javax.swing.*;
import java.util.*;
import java.text.NumberFormat;


/**
 * <p>
 * Agent representing the host for a party, to which a user-controlled number of guests
is invited.  The sequence is
 * as follows: the user selects a number guests to attend the party from 0 to 1000, using
the
 * slider on the UI.  When the party starts, the host creates N guest agents, each of
which registers
 * with the DF, and sends the host a message to say that they have arrived.  When all the
guests
 * have arrived, the party starts.  The host selects one guest at random, and tells them
a rumour.
 * The host then selects two other guests at random, and introduces them to each other.
```

```
The party
 * then proceeds as follows: each guest that is introduced to someone asks the host to
introduce them
 * to another guest (at random).  If a guest has someone introduce themselves, and the
guest knows
 * the rumour, they tell the other guest.  When a guest hears the rumour for the first
time, they
 * notify the host.  When all the guests have heard the rumour, the party ends and the
guests leave.
 * </p>
 * <p>
 * Note: to start the host agent, it must be named 'host'.  Thus:
 * <code><pre>
 *     java jade.Boot -gui host:com.hp.hpl.jade_test.HostAgent()
 * </pre></code>
 * </p>
 *
 * @author Ian Dickinson, HP Labs (<a href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */
public class HostAgent
    extends Agent
{
    // Constants
    //////////////////////////////////

    public final static String HELLO = "HELLO";
    public final static String ANSWER = "ANSWER";
    public final static String THANKS = "THANKS";
    public final static String GOODBYE = "GOODBYE";
    public final static String INTRODUCE = "INTRODUCE";
    public final static String RUMOUR = "RUMOUR";


    // Static variables
    //////////////////////////////////


    // Instance variables
    //////////////////////////////////
    protected JFrame m_frame = null;
    protected Vector m_guestList = new Vector();    // invitees
    protected int m_guestCount = 0;                 // arrivals
    protected int m_rumourCount = 0;
    protected int m_introductionCount = 0;
    protected boolean m_partyOver = false;
    protected NumberFormat m_avgFormat = NumberFormat.getInstance();
    protected long m_startTime = 0L;


    // Constructors
    //////////////////////////////////

    /**
     * Construct the host agent.  Some tweaking of the UI parameters.
     */
    public HostAgent() {
        m_avgFormat.setMaximumFractionDigits( 2 );
        m_avgFormat.setMinimumFractionDigits( 2 );
    }
```

```
    // External signature methods
    ////////////////////////////////

    /**
     * Setup the agent.  Registers with the DF, and adds a behaviour to
     * process incoming messages.
     */
    protected void setup() {
        try {
            System.out.println( getLocalName() + " setting up");

            // create the agent descrption of itself
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName( getAID() );
            DFService.register( this, dfd );

            // add the GUI
            setupUI();

            // add a Behaviour to handle messages from guests
            addBehaviour( new CyclicBehaviour( this ) {
                            public void action() {
                                ACLMessage msg = receive();

                                if (msg != null) {
                                    if (HELLO.equals( msg.getContent() )) {
                                        // a guest has arrived
                                        m_guestCount++;
                                        setPartyState( "Inviting guests (" + m_guestCount
+ " have arrived)" );

                                        if (m_guestCount == m_guestList.size()) {
                                            System.out.println( "All guests have arrived,
starting conversation" );
                                            // all guests have arrived
                                            beginConversation();
                                        }
                                    }
                                    else if (RUMOUR.equals( msg.getContent() )) {
                                        // count the agents who have heard the rumour
                                        incrementRumourCount();
                                    }
                                    else if (msg.getPerformative() == ACLMessage.REQUEST
&&  INTRODUCE.equals( msg.getContent() )) {
                                        // an agent has requested an introduction
                                        doIntroduction( msg.getSender() );
                                    }
                                }
                                else {
                                    // if no message is arrived, block the behaviour
                                    block();
                                }
                            }
                        } );
        }
        catch (Exception e) {
            System.out.println( "Saw exception in HostAgent: " + e );
            e.printStackTrace();
        }
```

```
    }


    // Internal implementation methods
    //////////////////////////////////

    /**
     * Setup the UI, which means creating and showing the main frame.
     */
    private void setupUI() {
        m_frame = new HostUIFrame( this );

        m_frame.setSize( 400, 200 );
        m_frame.setLocation( 400, 400 );
        m_frame.setVisible( true );
        m_frame.validate();
    }


    /**
     * Invite a number of guests, as determined by the given parameter.  Clears old
     * state variables, then creates N guest agents.  A list of the agents is maintained,
     * so that the host can tell them all to leave at the end of the party.
     *
     * @param nGuests The number of guest agents to invite.
     */
    protected void inviteGuests( int nGuests ) {
        // remove any old state
        m_guestList.clear();
        m_guestCount = 0;
        m_rumourCount = 0;
        m_introductionCount = 0;
        m_partyOver = false;
        ((HostUIFrame) m_frame).lbl_numIntroductions.setText( "0" );
        ((HostUIFrame) m_frame).prog_rumourCount.setValue( 0 );
        ((HostUIFrame) m_frame).lbl_rumourAvg.setText( "0.0" );

        // notice the start time
        m_startTime = System.currentTimeMillis();

        // try first to ensure that the DF has finished deregistering the old guests
        if (checkDF()) {

            setPartyState( "Inviting guests" );

            try {
                for (int i = 0;  i < nGuests;  i++) {
                    // create a new agent
                    Agent guest = new GuestAgent();
                    guest.doStart( "guest_" + i );

                    // keep the guest's ID on a local list
                    m_guestList.add( guest.getAID() );
                }
            }
            catch (Exception e) {
                System.err.println( "Exception while adding guests: " + e );
                e.printStackTrace();
            }
        }
```

```
    }


    /**
     * End the party: set the state variables, and tell all the guests to leave.
     */
    protected void endParty() {
        setPartyState( "Party over" );
        m_partyOver = true;

        // log the duration of the run
        System.out.println( "Simulation run complete. NGuests = " + m_guestCount + ",
time taken = " +
                            m_avgFormat.format( ((double) System.currentTimeMillis() -
m_startTime) / 1000.0 ) + "s" );

        // send a message to all guests to tell them to leave
        for (Iterator i = m_guestList.iterator();  i.hasNext();  ) {
            ACLMessage msg = new ACLMessage( ACLMessage.INFORM );
            msg.setContent( GOODBYE );

            msg.addReceiver( (AID) i.next() );

            send(msg);
        }

        m_guestList.clear();
    }


    /**
     * Shut down the host agent, including removing the UI and deregistering
     * from the DF.
     */
    protected void terminateHost() {
        try {
            if (!m_guestList.isEmpty()) {
                endParty();
            }

            DFService.deregister( this );
            m_frame.dispose();
            doDelete();
        }
        catch (Exception e) {
            System.err.println( "Saw FIPAException while terminating: " + e );
            e.printStackTrace();
        }
    }


    /**
     * Start the conversation in the party.  Tell a random guest a rumour, and
     * select two random guests and introduce them to each other.
     */
    protected void beginConversation() {
        // start a rumour
        ACLMessage rumour = new ACLMessage( ACLMessage.INFORM );
        rumour.setContent( RUMOUR );
        rumour.addReceiver( randomGuest( null ) );
        send( rumour );
```

```
        // introduce two agents to each other
        doIntroduction( randomGuest( null ) );
        setPartyState( "Swinging" );
    }


    /**
     * Introduce guest0 to a random other guest.  Also updates the introduction
     * count on the UI, and the avg no of introductions per rumour.
     */
    protected void doIntroduction( AID guest0 ) {
        if (!m_partyOver) {
            AID guest1 = randomGuest( guest0 );

            // introduce two guests to each other
            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( INTRODUCE + " " + guest0 );
            m.addReceiver( guest1 );
            send( m );

            // update the count of introductions on the UI
            m_introductionCount++;
            SwingUtilities.invokeLater( new Runnable() {
                                            public void run() {
                                                ((HostUIFrame)
m_frame).lbl_numIntroductions.setText( Integer.toString( m_introductionCount ));
                                            }
                                        } );
            updateRumourAvg();
        }
    }


    /**
     * Increment the number of guests that have heard the rumour, and update the UI.
     * If all guests have heard the rumour, end the party.
     */
    protected void incrementRumourCount() {
        m_rumourCount++;
        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).prog_rumourCount.setValue( Math.round( 100 * m_rumourCount / m_guestCount ) );
                                        }
                                    } );
        updateRumourAvg();

        // when all the guests have heard the rumour, the party ends
        if (m_rumourCount == m_guestCount) {
            // simulate the user clicking stop when the guests have all heard the rumour
            try {
                SwingUtilities.invokeAndWait( new Runnable() {
                                                  public void run() {
                                                      ((HostUIFrame)
m_frame).btn_stop_actionPerformed( null );
                                                  }
                                              } );
            }
            catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }
    }


    /**
     * Update the state of the party in the UI
     */
    protected void setPartyState( final String state ) {
        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).lbl_partyState.setText( state );
                                        }
                                    } );
    }


    /**
     * Update the average number of introductions per rumour spread
     * in the UI.
     */
    protected void updateRumourAvg() {
        SwingUtilities.invokeLater( new Runnable() {
                                        public void run() {
                                            ((HostUIFrame)
m_frame).lbl_rumourAvg.setText( m_avgFormat.format( ((double) m_introductionCount) /
m_rumourCount ) );
                                        }
                                    } );
    }


    /**
     * Pick a guest at random who is not the given guest.
     *
     * @param aGuest A guest at the party or null
     * @return A random guest who is not aGuest.
     */
    protected AID randomGuest( AID aGuest ) {
        AID g = null;

        do {
            int i = (int) Math.round( Math.random() * (m_guestList.size() - 1) );
            g = (AID) m_guestList.get( i );
        } while (g == aGuest);

        return g;
    }


    /**
     * Answer true if the DF has cleared all guests.  Note: this approach does
     * not work at the moment.
     */
    protected boolean checkDF() {
        try {
            ServiceDescription sd = new ServiceDescription();
            sd.setType( "PartyGuest" );
            sd.setName( "GuestServiceDescription" );
            DFAgentDescription dfd = new DFAgentDescription();
```

```
            dfd.addServices( sd );

            DFAgentDescription[] agents = DFService.search( this, dfd );

            // if not clear, warn user
            if (agents.length > 0) {
                JOptionPane.showMessageDialog( m_frame, "DF has not finished removing old
guest agents, please be patient", "Warning", JOptionPane.WARNING_MESSAGE );
            }
            else {
                return true;
            }
        }
        catch (Exception e) {
            System.err.println( "Exception: " + e );
            e.printStackTrace();
        }

        return false;
    }


    //============================================================================
    // Inner class definitions
    //============================================================================

}
```

```
/*****************************************************************************
 * Source code information
 * -----------------------
 * Original author    Ian Dickinson, HP Labs Bristol
 * Author email       Ian_Dickinson@hp.com
 * Package
 * Created            1 Oct 2001
 * Filename           $RCSfile:  $
 * Revision           $Revision:  $
 * Release status     Experimental.  $State: $
 *
 * Last modified on   $Date:  $
 *               by   $Author: $
 *
 * Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
 *****************************************************************************/


// Package
///////////////
package com.hp.hpl.jade_test;


// Imports
///////////////
import java.awt.*;
import javax.swing.*;
import java.beans.*;
import javax.swing.event.*;
import java.awt.event.*;

import jade.core.behaviours.OneShotBehaviour;


/**
 * TODO: Class comment.
 *
 * @author Ian Dickinson, HP Labs (<a href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */
public class HostUIFrame
    extends JFrame
{
    // Constants
    //////////////////////////////////


    // Static variables
    //////////////////////////////////


    // Instance variables
    //////////////////////////////////

    BorderLayout borderLayout1 = new BorderLayout();
    JPanel pnl_main = new JPanel();
    JButton btn_Exit = new JButton();
    Component component3;
    JButton btn_stop = new JButton();
    Component component2;
    JButton btn_start = new JButton();
    Box box_buttons;
```

```
    JPanel pnl_numGuests = new JPanel();
    BorderLayout borderLayout3 = new BorderLayout();
    JLabel lbl_numGuests = new JLabel();
    Box box_numGuests;
    JLabel lbl_guestCount = new JLabel();
    JSlider slide_numGuests = new JSlider();
    Component component1;
    Component component4;
    GridLayout gridLayout1 = new GridLayout();
    JLabel jLabel1 = new JLabel();
    JLabel jLabel2 = new JLabel();
    JLabel lbl_numIntroductions = new JLabel();
    JLabel jLabel4 = new JLabel();
    JLabel lbl_partyState = new JLabel();
    Box box1;
    JProgressBar prog_rumourCount = new JProgressBar();
    Component component6;
    Component component5;
    JLabel jLabel3 = new JLabel();
    JLabel lbl_rumourAvg = new JLabel();


    protected HostAgent m_owner;


    // Constructors
    ////////////////////////////////

    public HostUIFrame( HostAgent owner ) {
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }

        m_owner = owner;
    }


    // External signature methods
    ////////////////////////////////


    // Internal implementation methods
    ////////////////////////////////

    /**
     * Setup the UI. This code generated by JBuilder designer.
     */
    private void jbInit() throws Exception {
        component3 = Box.createHorizontalStrut(10);
        component2 = Box.createHorizontalStrut(5);
        box_buttons = Box.createHorizontalBox();

        box_numGuests = Box.createHorizontalBox();
        component1 = Box.createGlue();
        component4 = Box.createHorizontalStrut(5);
        box1 = Box.createVerticalBox();
        component6 = Box.createGlue();
        component5 = Box.createGlue();
```

```
        this.getContentPane().setLayout(borderLayout1);
        pnl_main.setLayout(gridLayout1);
        btn_Exit.setText("Exit");
        btn_Exit.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btn_Exit_actionPerformed(e);
            }
        });
        btn_stop.setEnabled(false);
        btn_stop.setText("Stop");
        btn_stop.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btn_stop_actionPerformed(e);
            }
        });
        btn_start.setText("Start");
        btn_start.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btn_start_actionPerformed(e);
            }
        });
        this.setTitle("Party Host Agent");
        this.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                this_windowClosing(e);
            }
        });
        pnl_numGuests.setLayout(borderLayout3);
        lbl_numGuests.setText("No. of guests:");
        lbl_guestCount.setMaximumSize(new Dimension(30, 17));
        lbl_guestCount.setMinimumSize(new Dimension(30, 17));
        lbl_guestCount.setPreferredSize(new Dimension(30, 17));
        lbl_guestCount.setText("10");
        slide_numGuests.setValue(10);
        slide_numGuests.setMaximum(1000);
        slide_numGuests.addChangeListener(new javax.swing.event.ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                slide_numGuests_stateChanged(e);
            }
        });
        gridLayout1.setRows(4);
        gridLayout1.setColumns(2);
        jLabel1.setToolTipText("");
        jLabel1.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel1.setText("Party state: ");
        jLabel2.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel2.setText("No. of introductions: ");
        lbl_numIntroductions.setBackground(Color.white);
        lbl_numIntroductions.setText("0");
        jLabel4.setToolTipText("");
        jLabel4.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel4.setText("Guests who have heard rumour: ");
        lbl_partyState.setBackground(Color.white);
        lbl_partyState.setText("Not started");
        prog_rumourCount.setForeground(new Color(0, 255, 128));
        prog_rumourCount.setStringPainted(true);
        jLabel3.setToolTipText("");
        jLabel3.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel3.setText("Avg. intros per rumour: ");
        lbl_rumourAvg.setToolTipText("");
        lbl_rumourAvg.setText("0.0");
```

```
        this.getContentPane().add(pnl_main, BorderLayout.CENTER);
        pnl_main.add(jLabel1, null);
        pnl_main.add(lbl_partyState, null);
        pnl_main.add(jLabel2, null);
        pnl_main.add(lbl_numIntroductions, null);
        pnl_main.add(jLabel4, null);
        pnl_main.add(box1, null);
        box1.add(component5, null);
        box1.add(prog_rumourCount, null);
        box1.add(component6, null);
        pnl_main.add(jLabel3, null);
        pnl_main.add(lbl_rumourAvg, null);
        this.getContentPane().add(pnl_numGuests, BorderLayout.NORTH);
        pnl_numGuests.add(box_numGuests, BorderLayout.CENTER);
        pnl_numGuests.setBorder( BorderFactory.createCompoundBorder(
BorderFactory.createEtchedBorder(), BorderFactory.createEmptyBorder( 2, 2, 2, 2 ) ) );

        box_numGuests.add(lbl_numGuests, null);
        box_numGuests.add(slide_numGuests, null);
        box_numGuests.add(lbl_guestCount, null);
        this.getContentPane().add(box_buttons, BorderLayout.SOUTH);
        box_buttons.add(component2, null);
        box_buttons.add(btn_start, null);
        box_buttons.add(component3, null);
        box_buttons.add(btn_stop, null);
        box_buttons.add(component1, null);
        box_buttons.add(btn_Exit, null);
        box_buttons.add(component4, null);
        lbl_partyState.setForeground( Color.black );
        lbl_numIntroductions.setForeground( Color.black );
        lbl_rumourAvg.setForeground( Color.black );
    }


    /**
     * When the slider for the num guests changes, we update the label.
     */
    void slide_numGuests_stateChanged(ChangeEvent e) {
        lbl_guestCount.setText( Integer.toString( slide_numGuests.getValue() ) );
    }


    /**
     * When the user clicks on start, notify the host to begin the party.
     */
    void btn_start_actionPerformed(ActionEvent e) {
        enableControls( true );

        // add a behaviour to the host to start the conversation going
        m_owner.addBehaviour( new OneShotBehaviour() {
                                  public void action() {
                                      ((HostAgent) myAgent).inviteGuests(
slide_numGuests.getValue() );
                                  }
                              } );
    }


    /**
     * When the user clicks on stop, tell the host to stop the party.
     */
```

```
    void btn_stop_actionPerformed(ActionEvent e) {
        enableControls( false );

        // add a behaviour to the host to end the party
        m_owner.addBehaviour( new OneShotBehaviour() {
                                  public void action() {
                                      ((HostAgent) myAgent).endParty();
                                  }
                              } );
    }


    /**
     * Maintains the enbabled/disabled state of key controls, depending
     * on whether the sim is running or stopped.
     */
    void enableControls( boolean starting ) {
        btn_start.setEnabled( !starting );
        btn_stop.setEnabled( starting );
        slide_numGuests.setEnabled( !starting );
        btn_Exit.setEnabled( !starting );
    }


    /**
     * When the user clicks the exit button, tell the host to shut down.
     */
    void btn_Exit_actionPerformed(ActionEvent e) {
        m_owner.addBehaviour( new OneShotBehaviour() {
                                  public void action() {
                                      ((HostAgent) myAgent).terminateHost();
                                  }
                              } );
    }


    /**
     * The window closing event is the same as clicking exit.
     */
    void this_windowClosing(WindowEvent e) {
        // simulate the user having clicked exit
        btn_Exit_actionPerformed( null );
    }


    //============================================================================
    // Inner class definitions
    //============================================================================

}
```

```
/*****************************************************************************
 * Source code information
 * -----------------------
 * Original author    Ian Dickinson, HP Labs Bristol
 * Author email       Ian_Dickinson@hp.com
 * Package
 * Created            1 Oct 2001
 * Filename           $RCSfile:  $
 * Revision           $Revision:  $
 * Release status     Experimental. $State: $
 *
 * Last modified on   $Date:  $
 *               by   $Author: $
 *
 * Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
 *****************************************************************************/


// Package
///////////////
package com.hp.hpl.jade_test;


// Imports
///////////////

import jade.core.Agent;
import jade.core.AID;

import jade.domain.FIPAException;

import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

import jade.core.behaviours.CyclicBehaviour;

import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.DFService;



/**
 * TODO: Class comment.
 *
 * @author Ian Dickinson, HP Labs (<a href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */
public class GuestAgent
    extends Agent
{
    // Constants
    //////////////////////////////////


    // Static variables
    //////////////////////////////////


    // Instance variables
    //////////////////////////////////
```

```
    protected boolean m_knowRumour = false;


    // Constructors
    ///////////////////////////////////


    // External signature methods
    ///////////////////////////////////

    /**
     * Set up the agent. Register with the DF, and add a behaviour to process
     * incoming messages.  Also sends a message to the host to say that this
     * guest has arrived.
     */
    protected void setup() {
        try {
            // create the agent descrption of itself
            ServiceDescription sd = new ServiceDescription();
            sd.setType( "PartyGuest" );
            sd.setName( "GuestServiceDescription" );
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName( getAID() );
            dfd.addServices( sd );

            // register the description with the DF
            DFService.register( this, dfd );

            // notify the host that we have arrived
            ACLMessage hello = new ACLMessage( ACLMessage.INFORM );
            hello.setContent( HostAgent.HELLO );
            hello.addReceiver( new AID( "host", false ) );
            send( hello );

            // add a Behaviour to process incoming messages
            addBehaviour( new CyclicBehaviour( this ) {
                            public void action() {
                                // listen if a greetings message arrives
                                ACLMessage msg = receive(
MessageTemplate.MatchPerformative( ACLMessage.INFORM ) );

                                if (msg != null) {
                                    if (HostAgent.GOODBYE.equals( msg.getContent() )) {
                                        // time to go
                                        leaveParty();
                                    }
                                    else if (msg.getContent().startsWith(
HostAgent.INTRODUCE )) {
                                        // I am being introduced to another guest
                                        introducing( msg.getContent().substring(
msg.getContent().indexOf( " " ) ) );
                                    }
                                    else if (msg.getContent().startsWith( HostAgent.HELLO
)) {
                                        // someone saying hello
                                        passRumour( msg.getSender() );
                                    }
                                    else if (msg.getContent().startsWith(
HostAgent.RUMOUR )) {
                                        // someone passing a rumour to me
                                        hearRumour();
```

```
                                    }
                                    else {
                                        System.out.println( "Guest received unexpected
message: " + msg );
                                    }
                                }
                                else {
                                    // if no message is arrived, block the behaviour
                                    block();
                                }
                            }
                        } );
        }
        catch (Exception e) {
            System.out.println( "Saw exception in GuestAgent: " + e );
            e.printStackTrace();
        }

    }


    // Internal implementation methods
    ////////////////////////////////

    /**
     * To leave the party, we deregister with the DF and delete the agent from
     * the platform.
     */
    protected void leaveParty() {
        try {
            DFService.deregister( this );
            doDelete();
        }
        catch (FIPAException e) {
            System.err.println( "Saw FIPAException while leaving party: " + e );
            e.printStackTrace();
        }
    }


    /**
     * Host is introducing this guest to the named guest.  Say hello to the guest,
     * and ask the host for another introduction.
     *
     * @param agentName The string form of the AID of the other guest.
     */
    protected void introducing( String agentName ) {
        // get the AID of the guest and send them a hello message
        AID aID = new AID( agentName.substring( agentName.lastIndexOf( ' ' ) + 1,
agentName.indexOf( ')' )), true );

        ACLMessage m = new ACLMessage( ACLMessage.INFORM );
        m.setContent( HostAgent.HELLO );
        m.addReceiver( aID );

        send( m );

        // request another introduction from the host
        ACLMessage m1 = new ACLMessage( ACLMessage.REQUEST );
        m1.setContent( HostAgent.INTRODUCE );
        m1.addReceiver( new AID( "host", false ) );
```

```
        send( m1 );
    }


    /**
     * Pass the rumour to the named guest, if we know it.
     *
     * @param agent Another guest we will send the rumour message to, but only if we
     *              know the rumour already.
     */
    protected void passRumour( AID agent ) {
        if (m_knowRumour) {
            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( HostAgent.RUMOUR );
            m.addReceiver( agent );
            send( m );
        }
    }


    /**
     * Someone has told this agent the rumour, we tell the host that we now know it.
     */
    protected void hearRumour() {
        // if I hear the rumour for the first time, tell the host
        if (!m_knowRumour) {
            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( HostAgent.RUMOUR );
            m.addReceiver( new AID( "host", false ) );
            send( m );

            m_knowRumour = true;
        }
    }



    //============================================================================
    // Inner class definitions
    //============================================================================

}
```

The GUI for the Party Host Agent

This example shows how (one way) to connect a GUI to a JADE agent.

```
/*****************************************************************************
 * Source code information
 * -----------------------
 * Original author    Ian Dickinson, HP Labs Bristol
 * Author email       Ian_Dickinson@hp.com
 * Package
 * Created            1 Oct 2001
 * Filename           $RCSfile:  $
 * Revision           $Revision:  $
 * Release status     Experimental.  $State: $
 *
 * Last modified on   $Date:  $
 *               by   $Author: $
 *
 * Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
 *****************************************************************************/
// Package
///////////////
package com.hp.hpl.jade_test;

// Imports
///////////////

import java.awt.*;
import javax.swing.*;
import java.beans.*;
import javax.swing.event.*;
import java.awt.event.*;
import jade.core.behaviours.OneShotBehaviour;


/**
 * TODO: Class comment.
 *
 * @author Ian Dickinson, HP Labs (<a href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */
public class HostUIFrame
    extends JFrame
{
    // Instance variables

    BorderLayout borderLayout1 = new BorderLayout();
    JPanel pnl_main = new JPanel();
    JButton btn_Exit = new JButton();
    Component component3;
    JButton btn_stop = new JButton();
    Component component2;
    JButton btn_start = new JButton();
```

```
Box box_buttons;
JPanel pnl_numGuests = new JPanel();
BorderLayout borderLayout3 = new BorderLayout();
JLabel lbl_numGuests = new JLabel();
Box box_numGuests;
JLabel lbl_guestCount = new JLabel();
JSlider slide_numGuests = new JSlider();
Component component1;
Component component4;
GridLayout gridLayout1 = new GridLayout();
JLabel jLabel1 = new JLabel();
JLabel jLabel2 = new JLabel();
JLabel lbl_numIntroductions = new JLabel();
JLabel jLabel4 = new JLabel();
JLabel lbl_partyState = new JLabel();
Box box1;
JProgressBar prog_rumourCount = new JProgressBar();
Component component6;
Component component5;
JLabel jLabel3 = new JLabel();
JLabel lbl_rumourAvg = new JLabel();
```

```
protected HostAgent m_owner;

// Constructors
//////////////////////////////////

public HostUIFrame( HostAgent owner ) {
    try {
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    m_owner = owner;
}

// Internal implementation methods

//////////////////////////////////
/**
 * Setup the UI. This code generated by JBuilder designer.
*
 */
private void jbInit() throws Exception {
    component3 = Box.createHorizontalStrut(10);
  component2 = Box.createHorizontalStrut(5);
  box_buttons = Box.createHorizontalBox();
  box_numGuests = Box.createHorizontalBox()
  component1 = Box.createGlue();
```

```
        component4 = Box.createHorizontalStrut(5);
        box1 = Box.createVerticalBox();
        component6 = Box.createGlue();
        component5 = Box.createGlue();
        this.getContentPane().setLayout(borderLayout1);
        pnl_main.setLayout(gridLayout1);
        btn_Exit.setText("Exit");
          btn_Exit.addActionListener(new
java.awt.event.ActionListener() {
              public void actionPerformed(ActionEvent e) {
                  btn_Exit_actionPerformed(e);
              }
          });
        btn_stop.setEnabled(false);
        btn_stop.setText("Stop");
          btn_stop.addActionListener(new
java.awt.event.ActionListener() {
              public void actionPerformed(ActionEvent e) {
                  btn_stop_actionPerformed(e);
              }
          });
        btn_start.setText("Start");
          btn_start.addActionListener(new
java.awt.event.ActionListener() {
              public void actionPerformed(ActionEvent e) {
                  btn_start_actionPerformed(e);
              }
          });
         this.setTitle("Party Host Agent");
         this.addWindowListener(new java.awt.event.WindowAdapter() {
           public void windowClosing(WindowEvent e) {
               this_windowClosing(e);
           }
       });
        pnl_numGuests.setLayout(borderLayout3);
        lbl_numGuests.setText("No. of guests:");
        lbl_guestCount.setMaximumSize(new Dimension(30, 17));
        lbl_guestCount.setMinimumSize(new Dimension(30, 17));
        lbl_guestCount.setPreferredSize(new Dimension(30, 17));
        lbl_guestCount.setText("10");
        slide_numGuests.setValue(10);
        slide_numGuests.setMaximum(1000);
```

```
        slide_numGuests.addChangeListener(new javax.swing.event.ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                slide_numuests_stateChanged(e);
            }
        });

        gridLayout1.setRows(4);
        gridLayout1.setColumns(2);
        jLabel1.setToolTipText("");
        jLabel1.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel1.setText("Party state: ");
        jLabel2.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel2.setText("No. of introductions: ");
        lbl_numIntroductions.setBackground(Color.white);
        lbl_numIntroductions.setText("0");
        jLabel4.setToolTipText("");
        jLabel4.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel4.setText("Guests who have heard rumour: ");
        lbl_partyState.setBackground(Color.white);
        lbl_partyState.setText("Not started");
        prog_rumourCount.setForeground(new Color(0, 255, 128));
        prog_rumourCount.setStringPainted(true);
        jLabel3.setToolTipText("");
        jLabel3.setHorizontalAlignment(SwingConstants.RIGHT);
        jLabel3.setText("Avg. intros per rumour: ");
        lbl_rumourAvg.setToolTipText("");
        lbl_rumourAvg.setText("0.0");
        this.getContentPane().add(pnlin, BorderLayout.CENTER);
        pnl_main.add(jLabel1, null);
        pnl_main.add(lbl_partyState, null);
        pnl_main.add(jLabel2, null);
        pnl_main.add(lbl_numIntroductions, null);
        pnl_main.add(jLabel4, null);
        pnl_main.add(box1, null);
        box1.add(component5, null);
          box1.add(prog_rumourCount, null);
        box1.add(component6, null);
        pnl_main.add(jLabel3, null);
        pnl_main.add(lbl_rumourAvg, null);
        this.getContentPane().add(pnl_numGuests, BorderLayout.NORTH);
        pnl_numGuests.add(box_numGuests, BorderLayout.CENTER);
        pnl_numGuests.setBorder( BorderFactory.createCompoundBorder(
BorderFactory.createEtchedBorder(), BorderFactory.createEmptyBorder( 2, 2, 2, 2 ) )
);

        box_numGuests.add(lbl_numGuests, null);
        box_numGuests.add(slide_numGuests, null);
        box_numGuests.add(lbl_guestCount, null);
        this.getContentPane().add(box_buttons, BorderLayout.SOUTH);
        box_buttons.add(component2, null);
        box_buttons.add(btn_start, null);
        box_buttons.add(component3, null);
        box_buttons.add(btn_stop, null);
        box_buttons.add(component1, null);
```

```
        box_buttons.add(btn_Exit, null);
        box_buttons.add(component4, null);
        lbl_partyState.setForeground( Color.black );
        lbl_numIntroductions.setForeground( Color.black );
        lbl_rumourAvg.setForeground( Color.black );
    }

    /**
     * When the slider for the num guests changes, we update the
label.
     */
    void slide_numGuests_stateChanged(ChangeEvent e) {
        lbl_guestCount.setText( Integer.toString(
slide_numGuests.getValue() ) );
    }
    /**
     * When the user clicks on start, notify the host to begin the
party.
     */
    void btn_start_actionPerformed(ActionEvent e) {
        enableControls( true );

        // add a behaviour to the host to start the conversation
going

        m_owner.addBehaviour( new OneShotBehaviour() {
                                    public void action() {
                                        ((HostAgent)
myAgent).inviteGuests( slide_numGuests.getValue() );

                                    }
                                });

    }

    /**
     * When the user clicks on stop, tell the host to stop the
party.
     */
    void btn_stop_actionPerformed(ActionEvent e) {
        enableControls( false );

        // add a behaviour to the host to end the party
        m_owner.addBehaviour( new OneShotBehaviour() {
                                    public void action() {
```

```
                                        ((HostAgent)
myAgent).endParty();
                                    }
                                } );
    }
   /**
     * Maintains the enbabled/disabled state of key controls,
depending
     * on whether the sim is running or stopped.
     */
    void enableControls( boolean starting ) {
        btn_start.setEnabled( !starting );
        btn_stop.setEnabled( starting );
        slide_numGuests.setEnabled( !starting );
        btn_Exit.setEnabled( !starting );
    }

    /**
     * When the user clicks the exit button, tell the host to shut
down.
     */
    void btn_Exit_actionPerformed(ActionEvent e)
        m_owner.addBehaviour( new OneShotBehaviour() {
                                    public void action() {
                                        ((HostAgent)
myAgent).terminateHost();
                                    }
                                } );
    }

    /**
     * The window closing event is the same as clicking exit.
     */
    void this_windowClosing(WindowEvent e) {
        // simulate the user having clicked exit
        btn_Exit_actionPerformed( null );
    }

}
```

Party Guest Agent

This agent is the party goer. The Host agent makes n copies of this and sets them to telling the rumour to one another.

```
/**************************************************************************
* Source code information
* -----------------------
* Original author Ian Dickinson, HP Labs Bristol
* Author email Ian_Dickinson@h.com
* Package
* Created 1 Oct 2001
* Filename $RCSfile: $
* Revision $Revision: $
* Release status Experimental. $State: $
*
* Last modified on $Date: $
* by $Author: $
*
* Copyright (c) 2001 Hewlett-Packard Company, all rights reserved.
**************************************************************************/
// Package
///////////////
package com.hp.hpl.jade_test;

// Imports
///////////////
import jade.core.Agent;
import jade.core.AID;
import jade.domain.FIPAException;
import jade.lang.acl.ACLMessage;
mport jade.lang.acl.MessageTemplate;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.DFService;

/**
 * TODO: Class comment.
 *
 * @author Ian Dickinson, HP Labs (<a
href="mailto:Ian_Dickinson@hp.com">email</a>)
 * @version CVS info: $Id: $
 */
public class GuestAgent
    extends Agent
{
```

```
    // Instance variables
    ////////////////////////////////

    protected boolean m_knowRumour = false;


    // External signature methods
                ////////////////////////////////
    /**
     * Set up the agent. Register with the DF, and add a behaviour
to process
     * incoming messages.  Also sends a message to the host to say
that this
     * guest has arrived.
     */

    protected void setup() {
        try {
            // create the agent descrption of itself
            ServiceDescription sd = new ServiceDescription();
            sd.setType( "PartyGuest" );
            sd.setName( "GuestServiceDescription" );
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName( getAID() );
            dfd.addServices( sd );

            // register the description with the DF
            DFService.register( this, dfd );

            // notify the host that we have arrived
            ACLMessage hello = new ACLMessage( ACLMessage.INFORM
);
            hello.setContent( HostAgent.HELLO );
            hello.addReceiver( new AID( "host", false );

            send( hello );

            // add a Behaviour to process incoming messages
            addBehaviour( new CyclicBehaviour( this ) {
                            public void action() {

                                // listen if a greetings message
arrives
                                ACLMessage msg = receive(
MessageTemplate.MatchPerformative( ACLMessage.INFORM ) );
```

```
                                if (msg != null) {
                                    if (HostAgent.GOODBYE.equals(
msg.getContent() )) {
                                        // time to go
                                        leaveParty();
                                    }
                                    else if
(msg.getContent().startsWith( HostAgent.INTRODUCE )) {
                                        // I am being introduced
to another guest
                                        introducing(
msg.getContent().substring( msg.getContent().indexOf( " " ) ) );
                                    }
                                    else if
(msg.getContent().startsWith( HostAgent.HELLO )) {
                                        // someone saying hello
                                        passRumour(
msg.getSender() );
                                    }
                                    else if
(msg.getContent().startsWith( HostAgent.RUMOUR )) {
                                        // someone passing a
rumour to me
                                        hearRumour();
                                    }
                                    else {
                                        System.out.println( "Guest
received unexpected message: " + msg );
                                    }
                                }
                                else {
                                    // if no message is arrived,
block the behaviour
                                    block();
                                }
                            }
                        } );
         }

        catch (Exception e) {
            System.out.println( "Saw exception in GuestAgent: " +
e );
            e.printStackTrace();
        }
```

```
    }

    // Internal implementation methods
    ////////////////////////////////
    /**
     * To leave the party, we deregister with the DF and delete
the agent from
     * the platform.
     */
    protected void leaveParty() {
        try {
            DFService.deregister( this );
            doDelete();
        }
        catch (FIPAException e) {
            System.err.println( "Saw FIPAException while leaving
party: " + e );
            e.printStackTrace();
        }
    }

    /**
     * Host is introducing this guest to the named guest.  Say
hello to the guest,
     * and ask the host for another introduction.
     *
     * @param agentName The string form of the AID of the other
guest.
     */
    protected void introducing( String agentName ) {
        // get the AID of the guest and send them a hello message
        AID aID = new AID( agentName.substring(
agentName.lastIndexOf( ' ' ) + 1, agentName.indexOf( ')' )), true
);

        ACLMessage m = new ACLMessage( ACLMessage.INFORM );
        m.setContent( HostAgent.HELLO );
        m.addReceiver( aID );

        send( m );

        // request another introduction from the host
        ACLMessage m1 = new ACLMessage( ACLMessage.REQUEST );
        m1.setContent( HostAgent.INTRODUCE );
        m1.addReceiver( new AID( "host", false ) );
```

```
        send( m1 );
    }

    /**
     * Pass the rumour to the named guest, if we know it.
     *
     * @param agent Another guest we will send the rumour message
to, but only if we
     *              know the rumour already.
     */
    protected void passRumour( AID agent ) {
        if (m_knowRumour) {
            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( HostAgent.RUMOUR );
            m.addReceiver( agent );

            send( m );
        }
    }
    /**
     * Someone has told this agent the rumour, we tell the host
that we now know it.
     */
    protected void hearRumour() {

        // if I hear the rumour for the first time, tell the host
        if (!m_knowRumour) {


            ACLMessage m = new ACLMessage( ACLMessage.INFORM );
            m.setContent( HostAgent.RUMOUR );
            m.addReceiver( new AID( "host", false ) );

            send( m );

            m_knowRumour = true;
        }
    }

}
```

Introduction to XML

XML stands for Extensible Markup Language. Like HTML it is derived from a more genral markup language called SGML, Standard General Markup Language. Both HTML and XML are defined in terms of SGML constructs.

Overview from Sun's Tutorial

What does XML look like? A configuration file for a web server (parsed by IE).

Glossary of XML terms

XML Alphabetic Index

XML is quite new and people see many possibilities for its use. The Sun Tutorial discusses some. Here is another view of the applicablility of XML.

- to describe metacontent of documents or on-line resources
- to publish and exchange database content
- to be a format for exchanging information between application programs (e.g., agents).

Here are a few points about each of these categories.

Metacontent of document and on-line resources

It is said that 92% of the information belonging to American corporations is stored in text documents. To computers, these are just strings of characters, or worse, pictures made up of pixels (after scanning, and without character recognition). To humans, these documents (all documents in fact) have structure which humans can (usually) see, and this structure helps humans to understand the documents.

In other words, humans interpret documents, aided by their structures. Machines cannot interpret these documents. In a sense, with regard to documents, computers are just glorified typewriters. Some people claim that the surprising failure of computers to increase the productivity of white collar workers is due to this failure of machine understanding of document structure. If computers hand some knowledge of this structure, more intelligent programs could be written to make use of documents by machines, as well as by people.

A markiup language puts tags in a text document to clarify its structure. The most well known markup language is HTML. HTML has a large set of tags which control the visual structure of a document, that is, the layout of the document. Clever layout allows humans to better understand the document. But layout understanding is not content understanding. HTML does not help structure the understanding of a document by machines.

Furthermore, HTML is standardized. Users cannot bend it to their own purposes. What is needed is a more flexible language, capable of structuring documents according to different critera, according to different semantics or ontologies. SGML has this capability but it is so large and complex that only a few gurus can use it. XML is a simplified version of SGML which can be used by mere mortals.

Because XML is extensible, it allows different documents to be structured in different ways, depending on the needs of readers, machine or human, as forseen by the person creating the markup.

Example.

Consider you want to search on the Web for articles or speeches by Bill Clinton. On the Web these documents are probably using HTML markups. That makes them look nice but for search purposes, HTML markups are not too useful. (Actually, you can put keywords in the header to help the search.) You will probably get thousands of hits, most of them "noise".

This search would be much more productive if there were an AUTHOR tag in HTML, but there isn't. What one needs is some kind of markup language which structures knowledge from a "library ontology". XML allows such a language to be created. XML is a language for creating custom markup languages.

Databases

Using HTML, data extracted from backend databases is ususally displayed in tables. This is rather rigid, and furthermore, the data is hard to manipulate because the table it is in just looks like a table, it is passive, not active. You can't do anything with it, other than look at it.

You could write a program using HTML's various TABLE tags to extract values from certain rows and colums and, say, add them. But a subsequent changes in layout could invalidate the calculation.

The trouble is that the content and the layout are mixed together. In the case of XML, XML concerns content structure, and the layout structure is given to another languages called XSL, Extensible Style Language. Meaning is separated from appearance.

Messaging

Here is the situation most related to Software Agents. XML has the potential to be a kind of "lingua franca", a universal language for communication among all kinds of agents, human, organizational (B2B - business to business), or artificial.

Many large businesses, such as banks, today use EDI (Electronic Data Interchange) for B2B commuications. This uses an arcane language called EDIFACT. These systems are expensive and hard to maintain. They are out of the reach for small and medium businesses. They are looking to the Internet.

Two problem and their solutions

- *Security*. Being solved by modern encryption techniques.
- *Agreement on a standard.* Here XML comes into the picture.

The DTD and the Parser

You might think that XML is too flexible. Everyone will create private languages. This is not the case because you can tell users how to use your private language. You send along with your XML text, a DTD, Document Type Definition. This instructs the user's XML parser how to interprert your language's structure. The user program can then have a program (often in Java) which uses the output of the parser.

Using XML with Java

The Sun tutorial shows two main ways of using XML with Java. Of course it uses the Sun XML parser.

The Java/XML Tutorial from Sun

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--

This file is the default configuration file for the
JSWDK server. Following is a brief overview of the
JSWDK server configuration options.

=======================================================

webserver dtd and xml:

Element
    Attribute(s)
    Element(s)
=======================================================

WebServer - A collection of web services managed by a
            single HTTP Web Server instance.

    id - A Unique Web Server id.
    adminPort - The Web Server administration port
                number which is used as an external
                Web Server hook to invoke administrative
                tasks such as gracefully shutting down
                the web server (note: these services
                are presently not specified and as such
                are subject to change).

    Service - A managed web service.

Service - A distinct web resource which is associated
          with a fully qualified URI.

    id - A unique Service id.
    port - The port number with which the Service is
           registered.
    hostName - The system host name in which the Service
               is hosted.
    inet - The system ip address in which the Service is
           hosted.
    docBase - The Service document base.
    workDir - The Service work directory.
    workDirIsPersistent - Indicator as to whether or not
                          the Web Server should return
                          the associated work directory
                          to the host system upon shut
                          down.

    WebApplication - A managed association of web resources.

WebApplication - A collection of associated web resources
                 which correspond to a distinct fully
                 qualified URI.

    id - A unique Web Application id.
    mapping - The URI prefix with which this Web Application
              is associated with relative to the hosted
              Service.
    docBase - The Web Application document base.
    maxInactiveInterval - The maximum session timeout
```

```
                        period.

=========================================================

command line options:

        -help                       This Message
        -config [file|url]          Read config from URL
        -noconfig                   Do not read config
        -adminport [int]            Administration Port
        -serviceid [str]            Service Id
        -port[:id|:*] [int]         Listen on int [for Service id]
        -inet[:id|:*] [inetaddr]    Bind server to inet [for Service id]
        -hostname[:id|:*] [name]    Use name as hostname [for Service id]
        -docbase[:id|:*] [name]     Use URL as the content base [for Service id]
        -workdir[:id|:*] [name]     Use scratch file [for Service id]

=========================================================

configuration attribute details:

    All "id" values must be unique with a collection
    of like elements.

    Most configuration values can be declared in one
    of following three means respectively:

        command line
        xml declaration
        default as specified by the dtd and/or
            application

    Many fields have default falues which are either
    specified in the associated dtd or within JSWDK.
    These default values can be changed by modifying
    the included webserver.xml and fully qualifying
    the appropriate element attributes.

    A JSWDK server can be started on any platform
    with no changes to the provided default
    webserver.xml configuration.

    The JSWDK server will create a default
    configuration file upon initialization if it does
    not find one either by looking in the JSWDK
    install directory for the file "webserver.xml or
    the "-config [file | url]" command line option.

    If an explicit WebServer.adminPort value is not
    specified then a series of 5 attempts will be
    made to bind to an available port number randomly
    chosen between the range of 2048 and 8192. Upon
    success, the chosen administration port is logged
    in the "webserverlog.txt" file. If an administration
    port cannot be determined, the web server will
    not be started.

    The WebServer.id field is likely to be removed
    in a future release.

    The Service.id field is required as the key for
```

```
    the command line arguments. Duplicate Server.port
    entries will cause the subsequent duplicate Services
    to be disregarded.

    The Server.hostName is opional and a value of
    "localhost" will be used if it remains
    unspecified.

    A Server.port field must be unique within a
    collection WebServer configuration. Subsequent
    Service instances specified with duplicate port
    numbers will fail initialization.

    The Server.docBase is the file system location
    which is accessed by the Web Server to route
    inbound http requests to a specific Service instance
    and not picked up by an associated Web Application.
    A document base can be specified as relative or
    absolute and need not reside within the JSWDK install
    directory. It should be possible to specify URI
    addresses as well effectively turning this WebServer
    Service instance into a proxy server although this is
    experimental with this release.

    The Server.workDir specfies the local file system
    directory available to the Web Server as needed to
    use as a cache, archive object persistence among
    other tasks. The work directory can be specified
    as relative or absolute and need not reside within
    the JSWDK install directory.

    The Server.workDirIsPersistent is an indicator to
    the WebServer to either save or return to the host
    system the associated work directory. This field is
    likely to be renamed to "isWorkDirPersistent" in a
    future release.

    The WebApplication.id field is likely to be
    removed in a future release.

    The WebApplication.mapping is used to specify the
    URI prefix with which this Web Application
    instance is to be associated with. The specified
    value of this field must be unique amongst a
    collection managed by a single Service instance.
    This field is likely to be renamed to "path" in a
    future release.

    The WebApplication.docBase is the file system
    location which is accessed by the Web Server to
    service inbound http requests routed to this
    specific Web Application instance. This field
    shares many of the attributes specified in the
    Server.docBase description above.

    The WebApplication.maxInactiveInterval is not
    utilized at this time and will likely specify
    the "session time out in minutes" threshold in
    a future release.

=========================================================
```

```
Miscellany:

    Any number of Service and/or Web Application
    instances can be readily added to an existing Web
    Server configuration by adding the appropriate and
    valid (as specified by the dtd and associated rules)
    xml details.

============================================================

-->

<!DOCTYPE WebServer [

<!ELEMENT WebServer (Service+)>
<!ATTLIST WebServer
    id ID #REQUIRED
    adminPort NMTOKEN "">

<!ELEMENT Service (WebApplication*)>
<!ATTLIST Service
    id ID #REQUIRED
    port NMTOKEN "8999"
    hostName NMTOKEN ""
    inet NMTOKEN ""
    docBase CDATA "webpages"
    workDir CDATA "work"
    workDirIsPersistent (false | true) "false">

<!ELEMENT WebApplication EMPTY>
<!ATTLIST WebApplication
    id ID #REQUIRED
    mapping CDATA #REQUIRED
    docBase CDATA #REQUIRED
    maxInactiveInterval NMTOKEN "30">
]>

<WebServer id="webServer">
    <Service id="service0">
        <WebApplication id="examples" mapping="/examples" docBase="examples"/>
    </Service>
         <Service id="service1">
                        <WebApplication id="cps840a4" mapping="/cps840a4"
docBase="cps840a4/WEB-INF/servlets"/>
         </Service>
</WebServer>
```

Using XML

The following example illustrates how XML is created and used.

First the designer decides how to structure a document bases on its semantics and ontology. In this example, we have an employee database ontology.

An XML Source File

File department.xml:

```
<?xml version="1.0"?>
<!DOCTYPE department SYSTEM "department.dtd">
<department>
  <employee id="J.D">
    <name>John Doe</name>
    <email>John.Doe@foo.ibm.com</email>
  </employee>
  <employee id="B.S">
    <name>Bob Smith</name>
    <email>Bob.Smith@foo.com</email>
  </employee>
  <employee id="A.M">
    <name>Alice Miller</name>
    <url href="http://www.trl.jp.ibm.com/~amiller/"/>
  </employee>
</department>
```

After roughing this out, perhaps with examples, it must be made precise using a DTD, here callled department.dtd. Here it is.

A DTD

File: department.dtd:

```
<!ELEMENT department (employee)*>
<!ELEMENT employee (name, (email | url))>
<!ATTLIST employee id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url EMPTY>
<!ATTLIST url href CDATA #REQUIRED>
```

The meaning of some of these terms will be discussed later. If the parser is 'non-validating', you do not actually need a DTD.

The Parser

To use this code, an XML parser is necessary. There are many free XML parsers. IBM and Microsoft both supply them. In fact Internet Explorer 5 has an XML parser built in. You can see this by looking at department.xml with IE5.

department.xml

This just makes the original code look pretty. But clearly IE5 "understands" something about XML. Try clicking department.xml with Netscape 4.6 to see the difference.

The DOM

The standard XML parser outputs a data structure called a Document Object Model (DOM). This is a tree structure. For the above example, the tree looks like this:

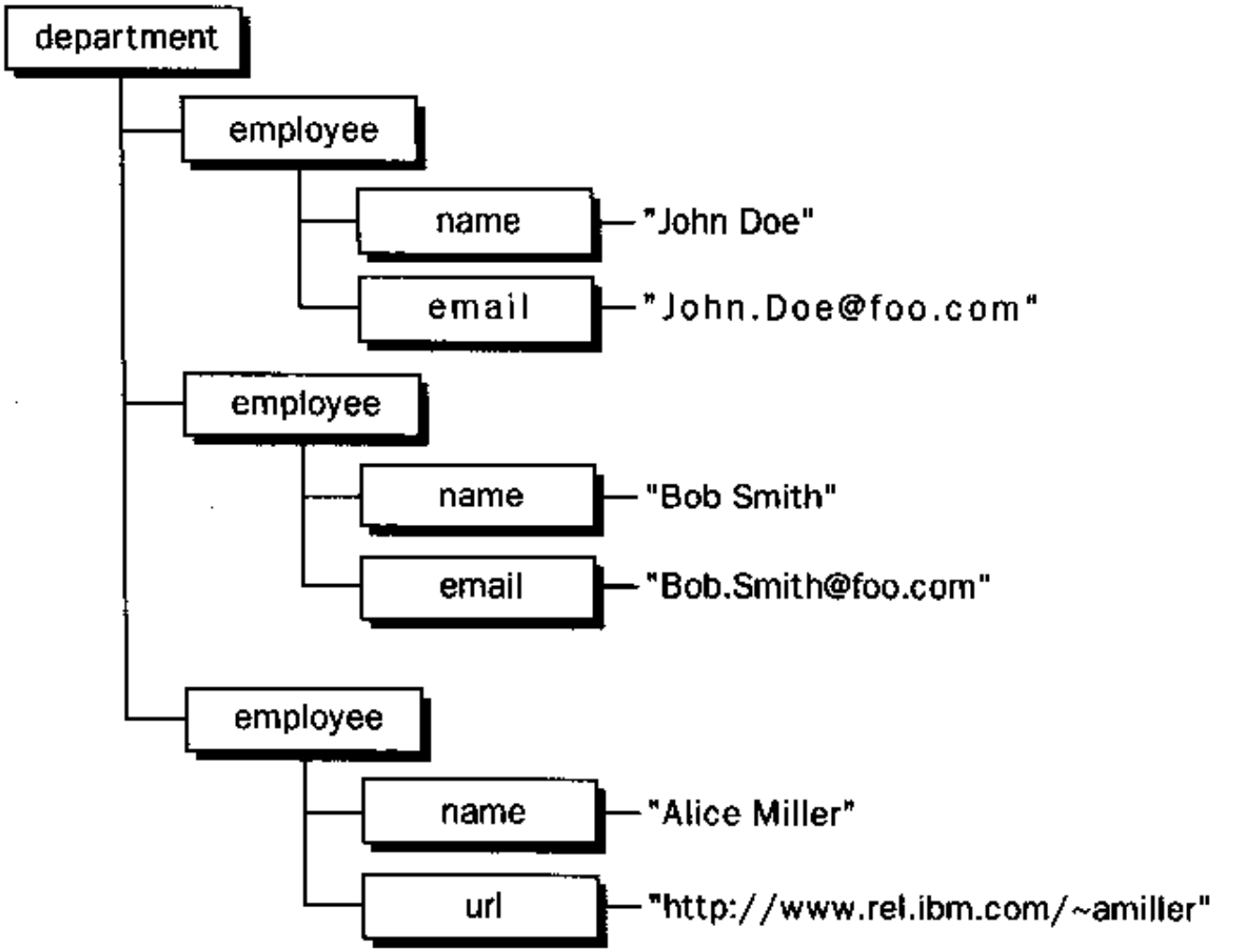


FIGURE 2.2 Structure of an XML document in DOM (`department.xml`)

The boxes are inner nodes called elements. The leafs of the treee are strings (called CDATA in XML).

SAX Parsers

There is a second kind of parser called tthe "Simple API for XML" or SAX for short. It does not build a tree. Instead, each element generates an event which the intepreting program can react to. The interpreting program can execute various callback methods in response to these events. The process is similar to responding to, say, ActionEvents using an ActionListener and the callback method actionPerformed.

In the case of SAX, you register interest in certain events with the parser and it calls back appropriate methods in your code when the events of interest occur.

SAX parsers are lightweight and do not need to store the wholle XML document in memory. The drawback is that the elements "whiz by" as the parse goes on. They generate their events and then are gone. You have to do your thing with the document in one pass. The DOM on the other had sits there once it is generated and the interperator program can interact with it at leisure.

The DOM method is more useful for programs interacting with a user. On the other hand, SAX is more useful for data exchange between systems, for example, for agent communication.

Validating Parsers

All XML parsers check the syntax of your XML file. They are much more strict than HTML parsers.

All parsers check for well-formed code. That is, for example, every opening tag has a closing tag. Validating parsers, in addition, try to check the logical structure as well. For example, a person cannot have two names. This is explicit in the DTD and a validating parser will catch an error such as this. Or, a tag not defined in the DTD could be present in a syntactically correct way. The validating parser will catch that too.

Interpretation

The final step in using XML is to write an interpreter of the ontology being represented in the DOM. Interpretation is meaning. The interpreter provides the semantics just as the parser represents the structure in its DOM output.

Java is a poplular language for writing such interpreters. Its "write once, run anywhere" philosophy is sybiotic to XML ambition to be a universal language of machine communication.

The "meaning" of the information represented in the XML-DTD-DOM resides in how the interpreter uses the DOM. The example above is not particularly exciting. The structure itself contains most of the meaning. Once could use the structure, to sort, or pick out individuals with certain properties, using efficient tree processing algorithms.

A More Complex XML Example

THe following example is developed in chapters 3 and 4 of Brett McLaughlen's book, *Java and XML*, (O'Reilly, 2000). The example describes the table of contents for the book.

contents.xml

```
<?xml version="1.0"?>
<?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
media="wap"?>
<?cocoon-process type="xslt"?>
<!DOCTYPE JavaXML:Book SYSTEM "DTD\JavaXML.dtd">
<!-- Java and XML -->
<JavaXML:Book xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/">
    <JavaXML:Title>Java and XML</JavaXML:Title>
    <JavaXML:Contents>
    <JavaXML:Chapter focus="XML">
        <JavaXML:Heading>Introduction</JavaXML:Heading>
        <JavaXML:Topic subSections="7">What Is It?</JavaXML:Topic>
        <JavaXML:Topic subSections="3">How Do I Use It?</JavaXML:Topic>
        <JavaXML:Topic subSections="4">Why Should I Use It?</JavaXML:Topic>
        <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
    </JavaXML:Chapter>
    <JavaXML:Chapter focus="XML">
        <JavaXML:Heading>Creating XML</JavaXML:Heading>
```

```
            <JavaXML:Topic subSections="0">An XML Document</JavaXML:Topic>
            <JavaXML:Topic subSections="2">The Header</JavaXML:Topic>
            <JavaXML:Topic subSections="6">The Content</JavaXML:Topic>
            <JavaXML:Topic subSections="1">What's Next?</JavaXML:Topic>
        </JavaXML:Chapter>
        <JavaXML:Chapter focus="Java">
            <JavaXML:Heading>Parsing XML</JavaXML:Heading>
            <JavaXML:Topic subSections="3">Getting Prepared</JavaXML:Topic>
            <JavaXML:Topic subSections="3">SAX Readers</JavaXML:Topic>
            <JavaXML:Topic subSections="9">Content Handlers</JavaXML:Topic>
            <JavaXML:Topic subSections="4">Error Handlers</JavaXML:Topic>
            <JavaXML:Topic subSections="0">A Better Way to Load a Parser</JavaXML:Topic>
            <JavaXML:Topic subSections="4">"Gotcha!"</JavaXML:Topic>
            <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
        </JavaXML:Chapter>
    <JavaXML:SectionBreak/>
        <JavaXML:Chapter focus="Java">
            <JavaXML:Heading>Web Publishing Frameworks</JavaXML:Heading>
            <JavaXML:Topic subSections="4">Selecting a Framework</JavaXML:Topic>
            <JavaXML:Topic subSections="4">Installation</JavaXML:Topic>
            <JavaXML:Topic subSections="3">Using a Publishing Framework</JavaXML:Topic>
            <JavaXML:Topic subSections="2">XSP</JavaXML:Topic>
            <JavaXML:Topic subSections="3">Cocoon 2.0 and Beyond</JavaXML:Topic>
            <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
        </JavaXML:Chapter>
    </JavaXML:Contents>
    <JavaXML:Copyright>&OReillyCopyright;</JavaXML:Copyright>
    </JavaXML:Book>
```

Attributes

In the above code there are a number of element attributes, "subSections", "focus", etc. xmlns:JavaXML is also an attribute.

There are several of these reserved attributes

- xml:lang
- xml:space
- xml:link

These are described in the XML Pocket Reference but are not on the course.

Attribute syntax

Attribute names cannot contain @, & or spaces. If they contain a ':' the part before it must be a namespace name. User defined attributes cannot start with xml.

Attributes are name/value pairs. The value is normally some kind of string.

Entity References

Entity reference are used for string substitutions. An entity reference begins with a '&' and ends with a ';'.

Avoiding the parser.

If you wish to use, say '<' , in data (PCDATA) then you can't do so directly because the parser will interpret it as the beginning of a tag. So you use < instead.

- <
- >
- &
- "
- '

You can also put in hex values this way. For example, the copyright symbol could be put in this way:

This document is © 1999, D Grimshaw.

Processing Instructions

<? target attribute="value", attribute="value" ... ?>

This is a processing instruction (PI). This information is passed from the XML file to the processing application. The programmer can develop her own. (not on cps720).

Note that there are some standard PI's built in. For example <?xml version="1.0" ?>

```
<?xml version="1.0"?>
<!DOCTYPE department SYSTEM "department.dtd">
<department>
  <employee id="J.D">
    <name>John Doe</name>
    <email>John.Doe@foo.ibm.com</email>
  </employee>

  <employee id="B.S">
    <name>Bob Smith</name>
    <email>Bob.Smith@foo.com</email>
  </employee>

  <employee id="A.M">
    <name>Alice Miller</name>
    <url href="http://www.trl.jp.ibm.com/~amiller/"/>
  </employee>
</department>
```

```
<?xml version="1.0"?>
<?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
                 media="wap"?>
<?cocoon-process type="xslt"?>
<!DOCTYPE JavaXML:Book SYSTEM "DTD\JavaXML.dtd">

<!-- Java and XML -->
<JavaXML:Book xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/">
 <JavaXML:Title>Java and XML</JavaXML:Title>
 <JavaXML:Contents>

  <JavaXML:Chapter focus="XML">
   <JavaXML:Heading>Introduction</JavaXML:Heading>
   <JavaXML:Topic subSections="7">What Is It?</JavaXML:Topic>
   <JavaXML:Topic subSections="3">How Do I Use It?</JavaXML:Topic>
   <JavaXML:Topic subSections="4">Why Should I Use It?</JavaXML:Topic>
   <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
  </JavaXML:Chapter>

  <JavaXML:Chapter focus="XML">
   <JavaXML:Heading>Creating XML</JavaXML:Heading>
   <JavaXML:Topic subSections="0">An XML Document</JavaXML:Topic>
   <JavaXML:Topic subSections="2">The Header</JavaXML:Topic>
   <JavaXML:Topic subSections="6">The Content</JavaXML:Topic>
   <JavaXML:Topic subSections="1">What's Next?</JavaXML:Topic>
  </JavaXML:Chapter>

  <JavaXML:Chapter focus="Java">
   <JavaXML:Heading>Parsing XML</JavaXML:Heading>
   <JavaXML:Topic subSections="3">Getting Prepared</JavaXML:Topic>
   <JavaXML:Topic subSections="3">SAX Readers</JavaXML:Topic>
   <JavaXML:Topic subSections="9">Content Handlers</JavaXML:Topic>
   <JavaXML:Topic subSections="4">Error Handlers</JavaXML:Topic>
   <JavaXML:Topic subSections="0">
     A Better Way to Load a Parser
   </JavaXML:Topic>
   <JavaXML:Topic subSections="4">"Gotcha!"</JavaXML:Topic>
   <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
  </JavaXML:Chapter>

  <JavaXML:SectionBreak/>

  <JavaXML:Chapter focus="Java">
   <JavaXML:Heading>Web Publishing Frameworks</JavaXML:Heading>
   <JavaXML:Topic subSections="4">Selecting a Framework</JavaXML:Topic>
   <JavaXML:Topic subSections="4">Installation</JavaXML:Topic>
   <JavaXML:Topic subSections="3">
     Using a Publishing Framework
   </JavaXML:Topic>
   <JavaXML:Topic subSections="2">XSP</JavaXML:Topic>
   <JavaXML:Topic subSections="3">Cocoon 2.0 and Beyond</JavaXML:Topic>
   <JavaXML:Topic subSections="0">What's Next?</JavaXML:Topic>
  </JavaXML:Chapter>

 </JavaXML:Contents>

 <JavaXML:Copyright>&OReillyCopyright;</JavaXML:Copyright>

</JavaXML:Book>
```

XML NameSpaces

JavaXML in JavaXML:Book etc is a namespace. Small, local xml documents do not need separate name spaces. But if a DTD and corresponding xml document is to be widely used on the Internet, steps must be taken to prevent name clashes. Such clashes might occur when two xml documents from different sources were combined into one document.

<JavaXML:Book xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/">

is a pointer to the owner of the name space.

xmlns: stands for xml name space. It is a reserved name, as are all names beginning with xml. To the right of the ':' is a *unique* identifier. A url does just fine.

You can have more than one name space in the same xml document. See page 8 of the XML Pocket Reference.

The Document Type Definition (DTD)

The syntax of the DTD is taken from SGML. The DTD is used to control the XML parser. This page details some DTD basic constructs.

DTD Placement

DTDs can be placed in a separate file, or at the beginning of an xml file.

In a separate file.

In this case the xml file begins like this,

```
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE department SYSTEM "department.dtd">
```

At the beginning of the xml file.

In this case the the DTD is enclosed withing [and], like this:

```
<!DOCTYPE department [
    <!-- definitions in here -->
]>
<!-- the xml document itself goes here -->
```

An example from Sun's WebServer config file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- the DTD -->
<!DOCTYPE WebServer [
<!ELEMENT WebServer (Service+)>
<!ATTLIST WebServer
    id ID #REQUIRED
    adminPort NMTOKEN "">
<!ELEMENT Service (WebApplication*)>
<!ATTLIST Service
    id ID #REQUIRED
    port NMTOKEN "8080"
    hostName NMTOKEN ""
    inet NMTOKEN ""
    docBase CDATA "webpages"
    workDir CDATA "work"
    workDirIsPersistent (false | true) "false">
<!ELEMENT WebApplication EMPTY>
<!ATTLIST WebApplication
    id ID #REQUIRED
    mapping CDATA #REQUIRED
    docBase CDATA #REQUIRED
    maxInactiveInterval NMTOKEN "30">
```

```
]>
<!-- the xml -->
<WebServer id="webServer">
    <Service id="service0">
        <WebApplication id="examples" mapping="/examples" docBase="examples"/>
    </Service>
         <Service id="service1">
                        <WebApplication id="cps840a4" mapping="/cps840a4"
docBase="cps840a4/WEB-INF/servlets"/>
         </Service>
</WebServer>
```

Some key words

(Note that xml is case sensitive.)

- DOCTYPE. The root node, naming the type of xml
- ELEMENT A non-leaf node in the tree. The basic element of the DOM
- ATTLIST Nodes can have attributes.
- CDATA Unparsed character data (i.e., strings. text uniterpreted by the parser).
- EMPTY The node does not contain data. (But has attributes ot be of any use.)
- ANY The node can contain anything.
- ID An identifier which must be unique for that element.
- IDREF A reference to an ID
- NMTOKENA valid XML name composed of letters, numbers, hyphens, underscores, and colons.
- #REQUIRED Must be present. An alternative is #IMPLIED

Some borrowings from BNF

- * zero or more
- + one or more
- ? zero or one
- | or

Some other syntactic details

- Lists of are enclosed in parentheses.
- Note the equal sign for attribute values.
- In the line, `workDirIsPersistent (false | true) "false", "`false" is the default. Several other lines in the example also have defaults.
- All character data must be enclosed in quotes.
- All element tags must be terminated. Empty elements are terminated with />.
- Each element lists its child nodes after its name.
- Of course, there is much more to the complete xml/DTD syntax.

Notes

- DTD language describes element trees, with the DOCTYPE as root. (Be able to draw such trees :-)).
- The elements in the above example contain no text data, just attributes.

Another Example

This example will be used later wrhen we look at interpreting DOMs.

averagegpa.xml

```
<?xml version="1.0"?>
<!-- test xml page -->
<!DOCTYPE averagegpa SYSTEM "averagegpa.dtd">
<averagegpa>
        <student>
                <firstname> Mary </firstname>
                <lastname> Wong </lastname>
                <sn> 97123456 </sn>
                <gpa> 4.01 </gpa>
                <grade Grade="A+" />
        </student>
        <student>
                <firstname> Brian </firstname>
                <lastname> Mulroney </lastname>
                <sn> 579874562 </sn>
                <gpa> 2.02</gpa>
                <grade Grade="C-" />
        </student>
        <student>
                <lastname> Kennedy </lastname>
                <sn> 763245610 </sn>
                <gpa> 2.78 </gpa>
                <grade Grade="B-" />
        </student>
</averagegpa>
```

A possible DTD to make this xml text "legal" to a parser is,

averagegpa.dtd

```
<!ELEMENT averagegpa (student)*>
<!ELEMENT student (firstname?, lastname, sn,gpa,grade)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT sn (#PCDATA) >
<!ELEMENT gpa (#PCDATA)>
<!ELEMENT grade EMPTY>
<!ATTLIST grade Grade CDATA #IMPLIED>
```

This very simple example illustrates elements (nodes) which contain text data. This contrasts with the nodes in the webserver example, which contain nothing but attributes.

Nodes which contain text are leaf nodes. The text is represented in the DTD by the symbol #PCDATA which stands for parsable character data.

Part of the tree represented by this DTD looks like this:

averagegpa
    student
        firstname
            "Mary"
        lastname
            "Wong"
        sn
            "97123456"
        gpa
            "4.01"
        grade Grade="A+"

Note that the values "Mary", "Wong". "97123456", and "4.01" are contained in nodes of type PCDATA. They are leaf nodes of the tree. The node, grade Grade="A+", is also a leaf node, but not of a PCDATA type, and it is one level higher in the tree than the other leafs.

A DTD for the the Table of Contents Example

Recall the XML file for the example from McLaughlin's *Java and XML*. Here is the corresponding DTD.

JavaXML.dtd

```
<!ELEMENT JavaXML:Book (JavaXML:Title, JavaXML:Contents, JavaXML:Copyright)>
<!ATTLIST JavaXML:Book xmlns:JavaXML CDATA #REQUIRED>
<!ELEMENT JavaXML:Title (#PCDATA)>
<!ELEMENT JavaXML:Contents ((JavaXML:Chapter+) (JavaXML:Chapter+,
JavaXML:SectionBreak?)+)>
<!ELEMENT JavaXML:Chapter (JavaXML:Heading?,JavaXML:Topic+)>
<!ATTLIST JavaXML:Chapter focus (XML|Java) "Java">
<!ELEMENT JavaXML:Heading (#PCDATA)>
<!ELEMENT JavaXML:Topic (#PCDATA)>
<!ATTLIST JavaXML:Topic subSections CDATA #IMPLIED>
<!ELEMENT JavaXML:SectionBreak EMPTY>
<!ELEMENT JavaXML:Copyright (#PCDATA)>
<!ENTITY OReillyCopyright SYSTEM
"http://www.oreilly.com/catalog/javaxml/docs/copyright.xml">
```

Things to note

- The use of the namespace JavaXML is made compulsory.
- The modifiers * (0 or more), + (one or more) and ? (0 or 1).

- The '|' which means exclusive or (xor).
- #REQUIRED AND #IMPLIED. The latter means optional. There is also #FIXED.

ENTITY

Entities are rather like macros. It allows you to substitute characters for other characters.

General Entities

<!ENTITY name "replacement characters">

These are defined in DTD's and used in xml documents.

For example,

<!ENTITY dg "David Grimsahaw">

which would allow you to have something like this in an xml file,

<NAME> &dg; </NAME>

You have already seen predefined entities such as < for '<<.

External Entities

This type allows you to copy other xml documents into yours. The above book index example has an example in its last line. In the corresponding xml file we have,

<JavaXML:Copyright>&OReillyCopyright;</JavaXML:Copyright>

Parameter Entities

This type is only for use withing DTD's. THe syntax is a bit different:

<!ENTITY % name "replacement characters">

Example

This is legal:

<!ENTITY % pcdata "(#PCDATA)">

<!ELEMENT author %pcdata>

For more details on DTD's, check the XML Pocket Reference.

```
<?xml version="1.0"?>
<!-- test xml page -->
<!DOCTYPE averagegpa SYSTEM "averagegpa.dtd">
<averagegpa>
        <student>
                <firstname> Mary </firstname>
                <lastname> Wong </lastname>
                <sn> 97123456 </sn>
                <gpa> 4.01 </gpa>
                <grade Grade="A+" />
        </student>
        <student>
                <firstname> Brian William</firstname>
                <lastname> Mulroney </lastname>
                <sn> 579874562 </sn>
                <gpa> 2.02</gpa>
                <grade Grade="C-" />
        </student>
        <student>
                <lastname> Kennedy </lastname>
                <sn> 763245610 </sn>
                <gpa> 2.78 </gpa>
                <grade Grade="B-" />
        </student>
</averagegpa>
```

```
<!ELEMENT averagegpa (student)*>
<!ELEMENT student (firstname?, lastname, sn,gpa,grade)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT sn (#PCDATA) >
<!ELEMENT gpa (#PCDATA)>
<!ELEMENT grade EMPTY>
<!ATTLIST grade Grade CDATA #IMPLIED>
```

```
<!ELEMENT JavaXML:Book (JavaXML:Title,
                        JavaXML:Contents,
                        JavaXML:Copyright)>
<!ATTLIST JavaXML:Book
      xmlns:JavaXML CDATA #REQUIRED
>
<!ELEMENT JavaXML:Title (#PCDATA)>
<!ELEMENT JavaXML:Contents ((JavaXML:Chapter+)|
                            (JavaXML:Chapter+, JavaXML:SectionBreak?)+)>
<!ELEMENT JavaXML:Chapter (JavaXML:Heading?,JavaXML:Topic+)>
<!ATTLIST JavaXML:Chapter
      focus (XML|Java) "Java"
>
<!ELEMENT JavaXML:Heading (#PCDATA)>
<!ELEMENT JavaXML:Topic (#PCDATA)>
<!ATTLIST JavaXML:Topic
      subSections CDATA #IMPLIED
>
<!ELEMENT JavaXML:SectionBreak EMPTY>
<!ELEMENT JavaXML:Copyright (#PCDATA)>
<!ENTITY OReillyCopyright SYSTEM
         "http://www.oreilly.com/catalog/javaxml/docs/copyright.xml">
```

Interpreting XML using a DOM Parser

Without an intepreter an XML document is meaningless. After all, you are defining your own "language" terms with your own tag names. The names remain meaningless without an interpreter.

Defining what meaning means is a difficult philisophical question! Or an AI question. To keep things simple, let's just say that the meaning of a text is revealed by the actions taken by its reader. In the case of the example, averagegpa.xml, humans can interpret the text, and therefore "know what it means". This text represents an ontology which we are used to and understand. We can take appropriate actions, perhaps just speech actions, upon reading it.

How can a machine "understand" this text? Like a human agent, it needs to know what actions can be taken given the ontology of the text. An interpretation program provides such an "understanding".

There are two types of XML parsers, SAX parsers and DOM parsers. The example on this page is a DOM parser. This parser first creates a Document Object Model (DOM). The DOM is a tree structure repersenting the whole XML document.

A very simple example.

[The xml and dtd files]

BestGPA2.java

This program interprets only one part of the xml file, the gpa element. It interprets it the way a human agent would, and uses its interpretation to decide which student (just strings to the computer!) is the best student. Or rather, the <student> element with the highers gpa value. This simple interpreter by no means has the richness of understanding of a human agent.

This program was written using IBM's XML Parser.

```
import com.ibm.xml.parser.Parser;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Hashtable;
import org.w3c.dom.CDATASection;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.EntityReference;
import org.w3c.dom.Node;
import org.w3c.dom.Text;
```

```
public class BestGPA2 {

    static public void main(String[] argv) {
        if (argv.length != 1) {
            System.err.println("Missing XML filename.");
            System.exit(1);
        }
        try {
            // Open specified file
            InputStream is = new FileInputStream(argv[0]);

            // Start parsing
            Parser parser = new Parser(argv[0]); // @XML4J
            Document doc = parser.readStream(is); // @XML4J

            // Check if there is errors
            if (parser.getNumberOfErrors() > 0) { // @XML4J
                System.exit(1);
            }
            // Document is well-formed

            float currentGPA = 0.0f, bestGPA = 0.0f;
            Student currentStudent = new Student();
            Student bestStudent = new Student();

            // The Document Element is <averagegpa> ... </averagegpa>
            // cycle through the <student> ... </student> elements
            for(Node student = doc.getDocumentElement().getFirstChild();
                student != null;
                student = student.getNextSibling()) {
                currentStudent.reset();
                if(student instanceof Element) {
                    for(Node data = student.getFirstChild();
                        data != null;
                        data = data.getNextSibling()) {
                        if(data.getNodeName().equals("firstname")) {
                            currentStudent.firstName =
                            data.getFirstChild().getNodeValue();
```

```
                    }
                    if(data.getNodeName().equals("lastname")) {
                        currentStudent.lastName =
                        data.getFirstChild().getNodeValue();
                    }
                    if(data.getNodeName().equals("sn")) {
                        currentStudent.studentNumber =
                        data.getFirstChild().getNodeValue();
                    }
                    if(data.getNodeName().equals("gpa")) {
                        currentStudent.gpa = (new
                        Float(getTheText(data))).floatValue();
                    }
                    if(data.getNodeName().equals("grade")) {
                    }

                } // end for_data
                currentStudent.displayStudent("Student");

                if(currentStudent.gpa > bestStudent.gpa) {
                    // (1) do NOT write bestStudent = currentSTudent; !!
                    // This makes currentStudent point to bestStudent.
                    // The result would be the best student would be filled
                    // by each student in turn. (best = last student!)
                    // (2) Field assignment is the most efficient
                    //bestStudent.firstName = currentStudent.firstName;
                    //bestStudent.gpa = currentStudent.gpa;
                    // (3) Cloning is neat, but it does involve the
                    // inefficiency of copying whole objects
                    bestStudent = (Student) currentStudent.clone();
                }
            } // end of if_student

        } // end of for_student
        bestStudent.displayStudent("Best Student");
    }
    catch (Exception e) {
        e.printStackTrace();
```

```
            }
        } // end main()

        /* An alternative using recursion. */
        /* ----------------------------------------------------*/
            private static String getTheText (Node node){
                // Create a StringBuffer to store the result.
                // StringBuffer is more efficient than String
                StringBuffer buffer = new StringBuffer();
                return getTheText1 (node, buffer);
            }

            private static String getTheText1 (Node node, StringBuffer buffer){

                // Visit all the child nodes
                for (Node ch = node.getFirstChild();
                    ch != null;
                    ch = ch.getNextSibling()) {
                // Recursively call if the child may have children
                if (ch instanceof Element || ch instanceof EntityReference) {
                    buffer.append(getTheText(ch));
                // If the child is a text, append it to the result buffer
                } else if (ch instanceof Text) {
                    buffer.append(ch.getNodeValue());
                }
                }
                return buffer.toString();
            }
        /* ---------------------------------------------------------- */
}

class Student extends Object implements Cloneable {

        // ugh! These should be private with getters and setters!
        public String firstName = null;
        public String lastName = null;
        public String studentNumber = null;
        public float gpa = 0.0f;
```

```
        public String grade = null;

        public Student() {
                gpa = 0.0f;
        }

        public void reset() {
                firstName = "";
                lastName = "";
                gpa = 0.0f;
                studentNumber = "";
                grade = "";
        }

        public Object clone() throws CloneNotSupportedException {
                return super.clone();
        }

        public void displayStudent(String title) {
                System.out.println("\n" + title + "\n");
                System.out.println(firstName + " " + lastName);
                System.out.println("Student number: " + studentNumber);
                System.out.println("Student GPA: " + gpa);
                System.out.println("Student letter grade: " + grade);
        }
}
```

The interesting features of this program are colour coded.

Some Actual XML Languages

Mathematical Markup Language

```
import com.ibm.xml.parser.Parser;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Hashtable;
import org.w3c.dom.CDATASection;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.EntityReference;
import org.w3c.dom.Node;
import org.w3c.dom.Text;

public class BestGPA2  {

static public void main(String[] argv) {
        if (argv.length != 1) {
                System.err.println("Missing XML filename.");
                System.exit(1);
        }
        try {
                // Open specified file
                InputStream is = new FileInputStream(argv[0]);

                // Start parsing
                Parser parser = new Parser(argv[0]);   // @XML4J
                Document doc = parser.readStream(is);  // @XML4J

                // Check if there is errors
                if (parser.getNumberOfErrors() > 0) {  // @XML4J
                        System.exit(1);
                }
                // Document is well-formed

                 float currentGPA = 0.0f, bestGPA = 0.0f;
                 Student currentStudent = new Student();
                 Student bestStudent = new Student();

                 // The Document Element is <averagegpa> ... </averagegpa>
                 // cycle through the <student> ... </student> elements
                 for(Node student = doc.getDocumentElement().getFirstChild();
                         student != null;
                         student = student.getNextSibling()) {
                        currentStudent.reset();
                        if(student instanceof Element) { // without this you get 2 of
everything!!
                                for(Node data = student.getFirstChild();
                                        data != null;
                                        data = data.getNextSibling()) {
                                        if(data.getNodeName().equals("firstname")) {
                                                currentStudent.firstName =
data.getFirstChild().getNodeValue();
                                        }
                                        if(data.getNodeName().equals("lastname")) {
                                                currentStudent.lastName =
data.getFirstChild().getNodeValue();
                                        }
                                        if(data.getNodeName().equals("sn")) {
                                                currentStudent.studentNumber =
data.getFirstChild().getNodeValue();
                                        }
                                        if(data.getNodeName().equals("gpa")) {
                                                currentStudent.gpa = (new
```

```
Float(getTheText(data))).floatValue();
                                        }
                                        if(data.getNodeName().equals("grade")) {
                                        }

                                } // end for_data
                                currentStudent.displayStudent("Student");

                                if(currentStudent.gpa > bestStudent.gpa) {
                                        // (1) do NOT write bestStudent = currentSTudent;
!!
                                        // This makes currentStudent point to
bestStudent.
                                        // The result would be the best student would be
filled
                                        // by each student in turn. (best = last
student!)
                                        // (2) Field assignment is the most efficient
                                        //bestStudent.firstName =
currentStudent.firstName;
                                        //bestStudent.gpa = currentStudent.gpa;
                                        // (3) Cloning is neat, but it does involve the
                                        // inefficiency of copying whole objects
                                        bestStudent = (Student) currentStudent.clone();
                                }
                  } // end of if_student

                } // end of for_student
                bestStudent.displayStudent("Best Student");
         }
         catch (Exception e) {
                 e.printStackTrace();
        }
} // end main()

/* ----------------------------------------------------------*/
private static String getTheText (Node node){
   // Create a StringBuffer to store the result.
   // StringBuffer is more efficient than String
   StringBuffer buffer = new StringBuffer();
   return getTheText1 (node, buffer);
}

private static String getTheText1 (Node node, StringBuffer buffer){

   // Visit all the child nodes
   for (Node ch = node.getFirstChild();
                ch != null;
                ch = ch.getNextSibling()) {
               // Recursively call if the child may have children
               if (ch instanceof Element || ch instanceof EntityReference) {
                       buffer.append(getTheText(ch));
               // If the child is a text, append it to the result buffer
               } else if (ch instanceof Text) {
                       buffer.append(ch.getNodeValue());
               }
   }
   return buffer.toString();
}
/* ------------------------------------------------------------------ */
}
```

```
class Student extends Object implements Cloneable {

public  String firstName = null;
public  String lastName = null;
public  String studentNumber = null;
public  float gpa = 0.0f;
public  String grade = null;

public Student() {
        gpa = 0.0f;
}

public void reset() {
        firstName = "";
        lastName = "";
        gpa = 0.0f;
        studentNumber = "";
        grade = "";
}

public Object clone() throws CloneNotSupportedException {
        return super.clone();
}

public void displayStudent(String title) {
        System.out.println("\n" + title + "\n");
        System.out.println(firstName + " " + lastName);
        System.out.println("Student number: " + studentNumber);
        System.out.println("Student GPA: " + gpa);
        System.out.println("Student letter grade: " + grade);
}
}
```

Using the SAX API

Overview

Using the SAX parser is a completely different experience from using a DOM parser. Programming SAX is event driven programming. Programming DOM is tree traversal.

Since SAX is event driven, programming SAX is not unlike programming the AWT or Swing. The SAX parser reads in an xml file (also a dtd if present) and each time it "sees" something interesting, such as an ELEMENT it generates a certain kind of event.

If your program is interested in this event, it can register with the parser as a listener by implementing certain interfaces. The SAX parser will then call back certain methods which you have overridden to do what you need to do in response to the event. Although the SAX parser organizes its events a bit differently, the situation is similar to an AWT Button generating ActionEvents when clicked. Then, an interested class implements an ActionListener and overrides the callback method actionPerformed.

SAX events, unlike AWT events, come in an ordered sequence as the xml file is read in. Given the tree structure of the xml file, the parser is generating events in a depth first order.

Furthermore, the events are fired "on the fly" as the xml file is read in. You only get one chance to "grab it" as it "flies" by. Unless you, the programmer, do something to capture relevant information at the point at which it arrives, the information is lost unless you parse the whole file again. This is in contrast to the DOM parser, which reads the who document in, storing it in its tree structure, and then waits for your program to analyze it at its leisure.

Example 1. A do nothing program.

Although this program does nothing it is actually quite useful! Remember that an xml file must be both well formed and valid. You can use this program to test an xml file for these properties. If the file is OK, then nothing apprears on stdout. On the other hand, if there is something wrong the program will throw an exception and print out an informative message.

CheckXML.java

```
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

import java.io.IOException;

/**
* Checks xml files for syntax errors. If a DTD is available, checks the validity of the xml
```

```
* file against the dtd.
*/
public class CheckXML {
        public static void main(String [] args) {
                if(args.length != 1) {
                        System.err.println("USAGE: java CheckXML <xml file name>");
                        System.exit(0);
                }
                try {
                        XMLReader parser = XMLReaderFactory.createXMLReader(
                        "org.apache.xerces.parsers.SAXParser");
                        parser.parse(args[0]);
                } catch (IOException e) {
                        System.out.println("Error reading URI: " + e.getMessage());
                } catch (SAXException e) {
                        System.out.println("Error in parsing: " + e.getMessage());
                }
        }
}
```

The code coloured in red shows how to set up the Xerces SAX parser.

Using SAX Callbacks

The four interfaces

The SAX API provides a number of interfaces which users can use to respond to events.

- ContentHandler (Legacy. Replaced by DocumentHandler in SAX 2.0.)
- DocumentHandler
- DTDHandler (Not on course.)
- ErrorHandler

Of these, the most important is the ContentHandler.

The next example illustrates using the callbacks provided by these interfaces.

Example 2. A program showing lots of events.

This example is taken from the O'Reilly book, *Java and XML* by Brett Mclaughlin.

JavaDoc documentation for SAXParserDemo.java

SAXParserDemo.java

```
/*--

Copyright (C) 2000 Brett McLaughlin. All rights reserved.

Redistribution and use in source and binary forms, with or without modifica-
tion, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions, the disclaimer that follows these conditions,
and/or other materials provided with the distribution.

3. Products derived from this software may not be called "Java and XML", nor may
"Java and XML" appear in their name, without prior written permission from
Brett McLaughlin (brett@newInstance.com).

THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
JDOM PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software was originally created by Brett McLaughlin <brett@newInstance.com>.
For more information on "Java and XML", please see <http://www.oreilly.com/catalog/javaxml/>
or <http://www.newInstance.com>.

*/
import java.io.IOException;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
```

```
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
* <b><code>SAXParserDemo</code></b> will take an XML file and parse it using SAX,
* displaying the callbacks in the parsing lifecycle.
*
* @author Brett McLaughlin
* @version 1.0
*/
public class SAXParserDemo {

/**
* <p>
* This parses the file, using registered SAX handlers, and output
* the events in the parsing process cycle.
* </p>
*
* @param uri <code>String</code> URI of file to parse.
*/
public void performDemo(String uri) {
    System.out.println("Parsing XML File: " + uri + "\n\n");

    // Get instances of our handlers
    ContentHandler contentHandler = new MyContentHandler();
    ErrorHandler errorHandler = new MyErrorHandler();

    try {
        // Instantiate a parser
        XMLReader parser =
        XMLReaderFactory.createXMLReader(
        "org.apache.xerces.parsers.SAXParser");

        // Register the content handler
        parser.setContentHandler(contentHandler);

        // Register the error handler
```

```
            parser.setErrorHandler(errorHandler);

            // Parse the document
            parser.parse(uri);

        } catch (IOException e) {
            System.out.println("Error reading URI: " + e.getMessage());
        } catch (SAXException e) {
            System.out.println("Error in parsing: " + e.getMessage());
        }

        }

/**
* <p>
* This provides a command line entry point for this demo.
* </p>
*/
public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java SAXParserDemo [XML URI]");
            System.exit(0);
        }

        String uri = args[0];

        SAXParserDemo parserDemo = new SAXParserDemo();
        parserDemo.performDemo(uri);
        }

}

/**
* <b><code>MyContentHandler</code></b> implements the SAX
* <code>ContentHandler</code> interface and defines callback
* behavior for the SAX callbacks associated with an XML
* document's content.
*/
```

```
class MyContentHandler implements ContentHandler {

    /** Hold onto the locator for location information */
    private Locator locator;

    /**
    * <p>
    * Provide reference to <code>Locator</code> which provides
    * information about where in a document callbacks occur.
    * </p>
    *
    * @param locator <code>Locator</code> object tied to callback
    * process
    */
    public void setDocumentLocator(Locator locator) {
        System.out.println(" * setDocumentLocator() called");
        // We save this for later use if desired.
        this.locator = locator;
    }

    /**
    * <p>
    * This indicates the start of a Document parse - this precedes
    * all callbacks in all SAX Handlers with the sole exception
    * of <code>{@link #setDocumentLocator}</code>.
    * </p>
    *
    * @throws <code>SAXException</code> when things go wrong
    */
    public void startDocument() throws SAXException {
        System.out.println("Parsing begins...");
    }

    /**
    * <p>
    * This indicates the end of a Document parse - this occurs after
    * all callbacks in all SAX Handlers.</code>.
```

```
        * </p>
        *
        * @throws <code>SAXException</code> when things go wrong
        */
        public void endDocument() throws SAXException {
            System.out.println("...Parsing ends.");
        }

        /**
        * <p>
        * This will indicate that a processing instruction (other than
        * the XML declaration) has been encountered.
        * </p>
        *
        * @param target <code>String</code> target of PI
        * @param data <code>String</code containing all data sent to the PI.
        * This typically looks like one or more attribute value
        * pairs.
        * @throws <code>SAXException</code> when things go wrong
        */
        public void processingInstruction(String target, String data)
        throws SAXException {

            System.out.println("PI: Target:" + target + " and Data:" + data);
        }

    /**
    * <p>
    * This will indicate the beginning of an XML Namespace prefix
    * mapping. Although this typically occur within the root element
    * of an XML document, it can occur at any point within the
    * document. Note that a prefix mapping on an element triggers
    * this callback <i>before</i> the callback for the actual element
    * itself (<code>{@link #startElement}</code>) occurs.
    * </p>
    *
    * @param prefix <code>String</code> prefix used for the namespace
```

```
* being reported
* @param uri <code>String</code> URI for the namespace
* being reported
* @throws <code>SAXException</code> when things go wrong
*/
public void startPrefixMapping(String prefix, String uri) {
        System.out.println("Mapping starts for prefix " + prefix +
        " mapped to URI " + uri);
}

/**
* <p>
* This indicates the end of a prefix mapping, when the namespace
* reported in a <code>{@link #startPrefixMapping}</code> callback
* is no longer available.
* </p>
*
* @param prefix <code>String</code> of namespace being reported
* @throws <code>SAXException</code> when things go wrong
*/
public void endPrefixMapping(String prefix) {
        System.out.println("Mapping ends for prefix " + prefix);
}

/**
* <p>
* This reports the occurrence of an actual element. It will include
* the element's attributes, with the exception of XML vocabulary
* specific attributes, such as
* <code>xmlns:[namespace prefix]</code> and
* <code>xsi:schemaLocation</code>.
* </p>
*
* @param namespaceURI <code>String</code> namespace URI this element
* is associated with, or an empty
* <code>String</code>
* @param localName <code>String</code> name of element (with no
```

```
* namespace prefix, if one is present)
* @param rawName <code>String</code> XML 1.0 version of element name:
* [namespace prefix]:[localName]
* @param atts <code>Attributes</code> list for this element
* @throws <code>SAXException</code> when things go wrong
*/
public void startElement(String namespaceURI, String localName,
String rawName, Attributes atts)
        throws SAXException {

        System.out.print("startElement: " + localName);
        if (!namespaceURI.equals("")) {
                System.out.println(" in namespace " + namespaceURI +
                " (" + rawName + ")");
        } else {
                System.out.println(" has no associated namespace");
        }

        for (int i=0; i<atts.getLength(); i++)
                System.out.println(" Attribute: " + atts.getLocalName(i) +
                "=" + atts.getValue(i));
}

/**
* <p>
* Indicates the end of an element
* (<code>&lt;/[element name]&gt;</code>) is reached. Note that
* the parser does not distinguish between empty
* elements and non-empty elements, so this will occur uniformly.
* </p>
*
* @param namespaceURI <code>String</code> URI of namespace this
* element is associated with
* @param localName <code>String</code> name of element without prefix
* @param rawName <code>String</code> name of element in XML 1.0 form
* @throws <code>SAXException</code> when things go wrong
*/
```

```
public void endElement(String namespaceURI, String localName,
        String rawName)
        throws SAXException {

        System.out.println("endElement: " + localName + "\n");
}

/**
* <p>
* This will report character data (within an element).
* </p>
*
* @param ch <code>char[]</code> character array with character data
* @param start <code>int</code> index in array where data starts.
* @param end <code>int</code> index in array where data ends.
* @throws <code>SAXException</code> when things go wrong
*/
public void characters(char[] ch, int start, int end)
        throws SAXException {

        String s = new String(ch, start, end);
        System.out.println("characters: " + s);
}

/**
* <p>
* This will report whitespace that can be ignored in the
* originating document. This is typically only invoked when
* validation is ocurring in the parsing process.
* </p>
*
* @param ch <code>char[]</code> character array with character data
* @param start <code>int</code> index in array where data starts.
* @param end <code>int</code> index in array where data ends.
* @throws <code>SAXException</code> when things go wrong
*/
public void ignorableWhitespace(char[] ch, int start, int end)
```

```
throws SAXException {

        String s = new String(ch, start, end);
        System.out.println("ignorableWhitespace: [" + s + "]");
}

/**
* <p>
* This will report an entity that is skipped by the parser. This
* should only occur for non-validating parsers, and then is still
* implementation-dependent behavior.
* </p>
*
* @param name <code>String</code> name of entity being skipped
* @throws <code>SAXException</code> when things go wrong
*/
public void skippedEntity(String name) throws SAXException {
        System.out.println("Skipping entity " + name);
}

}

/**
* <b><code>MyErrorHandler</code></b> implements the SAX
* <code>ErrorHandler</code> interface and defines callback
* behavior for the SAX callbacks associated with an XML
* document's errors.
*/
class MyErrorHandler implements ErrorHandler {

        /**
        * <p>
        * This will report a warning that has occurred; this indicates
        * that while no XML rules were "broken", something appears
        * to be incorrect or missing.
        * </p>
        *
        * @param exception <code>SAXParseException</code> that occurred.
```

```
* @throws <code>SAXException</code> when things go wrong
*/
public void warning(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Warning**\n" +
        " Line: " +
        exception.getLineNumber() + "\n" +
        " URI: " +
        exception.getSystemId() + "\n" +
        " Message: " +
        exception.getMessage());
        throw new SAXException("Warning encountered");
}

/**
* <p>
* This will report an error that has occurred; this indicates
* that a rule was broken, typically in validation, but that
* parsing can reasonably continue.
* </p>
*
* @param exception <code>SAXParseException</code> that occurred.
* @throws <code>SAXException</code> when things go wrong
*/
public void error(SAXParseException exception)
throws SAXException {

        System.out.println("**Parsing Error**\n" +
        " Line: " +
        exception.getLineNumber() + "\n" +
        " URI: " +
        exception.getSystemId() + "\n" +
        " Message: " +
        exception.getMessage());
        throw new SAXException("Error encountered");
}
```

```
        /**
        * <p>
        * This will report a fatal error that has occurred; this indicates
        * that a rule has been broken that makes continued parsing either
        * impossible or an almost certain waste of time.
        * </p>
        *
        * @param exception <code>SAXParseException</code> that occurred.
        * @throws <code>SAXException</code> when things go wrong
        */
        public void fatalError(SAXParseException exception)
        throws SAXException {

            System.out.println("**Parsing Fatal Error**\n" +
            " Line: " +
            exception.getLineNumber() + "\n" +
            " URI: " +
            exception.getSystemId() + "\n" +
            " Message: " +
            exception.getMessage());
            throw new SAXException("Fatal Error encountered");
        }

    }
```

Most of the useful work is done in the methods startDocument(), startElement(), endElement(), and characters() shown in red.

Because the two programmer defined classes implement interfaces directly, all the methods in the interfaces must be given some kind of body. This is inconvenient so the SAX provides some convenience classes to provide empty implementations of all these methods. This is just like the adapter classes such as WindowAdapter provided by the AWT.

HTML Doc for the SAXDemo program.

Example 3. Doing something with SAX

This example illustrates the use of an adapter class and also shows SAX extracting some data from a document and doing something with the data.

The adapter class is called DefaultHandler. Note that it is in the org.xml.sax.helper package not the org.xml.sax package. (DefaultHandler is a version 2.0 replacement for HandleBase in version 1.0. HandleBase performs a similar function. It is in org.xml.sax but has been deprecated.)

- averagegpa.dtd
- averagegpa.xml

GPAExample.java

```
package xml.sax.gpaExample2;

import java.io.IOException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
* A SAX parser example with a simple calculation.<p>
* The file to be parsed conforms to the DTD averagegpa.dtd. An example is
* averagegpa.xml. <p>
* The program finds the student with the highest gpa. Output is to stdout.<p>
*
* SAX parsers analyse the xml document dynamically, producing events for each type
* of XML data. The analysis proceeds in depth first order. The program must arrange
* to capture relevant data as it "flies by". <p>
*
* Various interfaces provide callback methods to respond to these events. SAX 2.0 provides
* the convenience class DefaultHandler which provides empty implementations off all
* the methods in these interfaces. You subclass this class and override the methods of
* your choosing to provide the desired functionality. <p>
*
* This program uses several of the most useful of these callbacks, characters(), startElement(),
* and endElement().<p>
```

```
*/
public class GPAExample {

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java GPAExample [XML URI]");
            System.exit(0);
        }

        String uri = args[0];

        GPAExample gpaAnalysis = new GPAExample();
        gpaAnalysis.analyse(uri);
    }

    /**
    * Sets up the parser. These calls are very standardized.
    * @param uri <code> String</code>The locator for the XML file to be parsed.
    */
    public void analyse(String uri) {

        // Get instances of our handlers. DefaultHandler is a convenience
        // class which implements default empty methods for4 interfaces,
        // EntityResolver, DTDHandler, ContentHandler and ErrorHandler.
        // Subclass and override the methods you need.

        DefaultHandler theHandler = new MyHandler(); // creae the subclass

        try {
            // Instantiate a parser
            XMLReader parser =
            XMLReaderFactory.createXMLReader(
            "org.apache.xerces.parsers.SAXParser");

            // Register the content handler (part of DefaultHandler)
            parser.setContentHandler(theHandler);

            // Register the error handler (Should be done sometime.)
```

```
                    //parser.setErrorHandler(errorHandler);

                    // Parse the document
                    parser.parse(uri);

                } catch (IOException e) {
                    System.out.println("Error reading URI: " + e.getMessage());
                } catch (SAXException e) {
                    System.out.println("Error in parsing: " + e.getMessage());
                }
        }

}
/**
* Tailors the DefaultHandler for this application.
*/
class MyHandler extends DefaultHandler {

        /*
        * The startElement() and endElement() methods are called every time the parser
        * sees an Element. These variables, global to the class, allow the different types
        * of elements to be distinguished across calls. In particular the control the action of
        * the characters() method which reads PCDATA from the XML file.
        */
        private boolean isGPA = false;
        private boolean isFirstName = false;
        private boolean isLastName = false;

        // retain some of the parsed data when the SAX parser moves on

        private Student bestStudent, aStudent;

        public MyHandler() {
                bestStudent = new Student("","", 0.00f, "");
        }

        public void startDocument() throws SAXException {
                System.out.println("Parsing begins...");
```

```
}

public void endDocument() throws SAXException {
        // Called at the end, so print out the result.
        System.out.println("The Best Student");
        System.out.println("================");
        bestStudent.printStudent();
        System.out.println("...Parsing ends.");
}

// This captures the PCDATA etc in an element.

public void characters(char[] ch, int start, int end)
throws SAXException {

        String s = new String(ch, start, end);

        // keep the parsed data. The aStudent variable should not be null because
        // it got the reference when startElement() was called on <student> tag.
        // Similarly the boolean flags are set in starElement().

        if(isFirstName) {
                if(aStudent != null) {
                        aStudent.setFirstName(s);
                }
        } else if(isLastName) {
                if(aStudent != null) {
                        aStudent.setLastName(s);
                }
        } else if(isGPA) {
                if(aStudent != null) {
                        float gp = (new Float(s)).floatValue();
                        aStudent.setGpa(gp);
                }
        }
}

public void startElement(String namespaceURI, String localName,
```

```
            String rawName, Attributes atts)
            throws SAXException {

            if(localName.equals("student")) {
                  aStudent = new Student("", "", 0.0f, "");
            }

            // Flags to control the characters() method.
            // Distinguish among types of elements.

            if(localName.equals("gpa")) {
                  isGPA = true;
            } else if(localName.equals("firstname")) {
                  isFirstName = true;
            } else if(localName.equals("lastname")) {
                  isLastName = true;
            }
      }

      public void endElement(String namespaceURI, String localName,
      String rawName)
      throws SAXException {

            // Check that the event signals the end of a <student> element. If so, the
            // current student's (aStudent) data fields are populated. So now you can
            // compare the current student to the best so far student and perhaps replace
            // the best so far student.

            if(localName.equals("student") && aStudent != null) {
                  if(aStudent.getGpa() > bestStudent.getGpa()) {
                        bestStudent = aStudent;
                  }
                  aStudent.printStudent();

                  // allow garbage collection of old student data
                  aStudent = null;
            }
            // At any time, two of these are already false, but it's a waste of time to test.
```

```
            isGPA = false;
            isFirstName = false;
            isLastName = false;
        }
        /**
        * A class to represent student data. The fields correspond to those in the XML
        document.
        */
        class Student {
            private String firstName;
            private String lastName;
            private float gpa;
            private String grade;

            public Student(String fn, String ln, float gpa, String g) {
                firstName = fn;
                lastName = ln;
                this.gpa = gpa;
                grade = g;
            }
            public void setFirstName(String fn) {
                firstName = fn;
            }
            public void setLastName(String ln) {
                lastName = ln;
            }
            public void setGpa(float g) {
                gpa = g;
            }
            public float getGpa() {
                return gpa;
            }
            public void setGrade(String g) {
                grade = g;
            }
            public void printStudent() {
                System.out.println(firstName);
```

```
                System.out.println(lastName);
                System.out.println(gpa);
                System.out.println(grade);
            }
        }
    }
```

[top] [previous] [next] Questions?

```
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

import java.io.IOException;

/**
 * Checks xml files for syntax errors. If a DTD is available, checks the validity of the
xml
 * file against the dtd.
 */
public class CheckXML {
        public static void main(String [] args) {
                if(args.length != 1) {
                        System.err.println("USAGE: java CheckXML <xml file name>");
                        System.exit(0);
                }
                try {
                        XMLReader parser = XMLReaderFactory.createXMLReader(

"org.apache.xerces.parsers.SAXParser");
                        parser.parse(args[0]);
                } catch (IOException e) {
                                System.out.println("Error reading URI: " +
e.getMessage());
                 } catch (SAXException e) {
                                System.out.println("Error in parsing: " +
e.getMessage());
        }
        }
}
```

All Classes

MyContentHandler
MyErrorHandler
SAXParserDemo

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class MyContentHandler

```
java.lang.Object
  |
  +--MyContentHandler
```

All Implemented Interfaces:

org.xml.sax.ContentHandler

class **MyContentHandler**

extends java.lang.Object

implements org.xml.sax.ContentHandler

`MyContentHandler` implements the SAX `ContentHandler` interface and defines callback behavior for the SAX callbacks associated with an XML document's content.

Constructor Summary

(package private)	**`MyContentHandler`**`()`

Method Summary

void	**`characters`**`(char[] ch, int start, int end)` This will report character data (within an element).
void	**`endDocument`**`()` This indicates the end of a Document parse - this occurs after all callbacks in all SAX Handlers..
void	**`endElement`**`(java.lang.String namespaceURI, java.lang.String localName, java.lang.String rawName)` Indicates the end of an element (`</[element name]>`) is reached.

void	**endPrefixMapping**(java.lang.String prefix) This indicates the end of a prefix mapping, when the namespace reported in a startPrefixMapping(java.lang.String, java.lang.String) callback is no longer available.
void	**ignorableWhitespace**(char[] ch, int start, int end) This will report whitespace that can be ignored in the originating document.
void	**processingInstruction**(java.lang.String target, java.lang.String data) This will indicate that a processing instruction (other than the XML declaration) has been encountered.
void	**setDocumentLocator**(org.xml.sax.Locator locator) Provide reference to Locator which provides information about where in a document callbacks occur.
void	**skippedEntity**(java.lang.String name) This will report an entity that is skipped by the parser.
void	**startDocument**() This indicates the start of a Document parse - this precedes all callbacks in all SAX Handlers with the sole exception of setDocumentLocator(org.xml.sax.Locator).
void	**startElement**(java.lang.String namespaceURI, java.lang.String localName, java.lang.String rawName, org.xml.sax.Attributes atts) This reports the occurrence of an actual element.
void	**startPrefixMapping**(java.lang.String prefix, java.lang.String uri) This will indicate the beginning of an XML Namespace prefix mapping.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MyContentHandler

`MyContentHandler``()`

Method Detail

setDocumentLocator

`public void` **`setDocumentLocator`**`(org.xml.sax.Locator locator)`

Provide reference to `Locator` which provides information about where in a document callbacks occur.

Specified by:

`setDocumentLocator` in interface `org.xml.sax.ContentHandler`

Parameters:

`locator` - `Locator` object tied to callback process

---

startDocument

`public void` **`startDocument`**`()`
`throws org.xml.sax.SAXException`

This indicates the start of a Document parse - this precedes all callbacks in all SAX Handlers with the sole exception of `setDocumentLocator(org.xml.sax.Locator)`.

Specified by:

`startDocument` in interface `org.xml.sax.ContentHandler`

Throws:

`SAXException` - when things go wrong

---

endDocument

`public void` **`endDocument`**`()`
`throws org.xml.sax.SAXException`

This indicates the end of a Document parse - this occurs after all callbacks in all SAX Handlers..

Specified by:

`endDocument` in interface `org.xml.sax.ContentHandler`

Throws:

`SAXException` - when things go wrong

---

processingInstruction

```
public void processingInstruction(java.lang.String target,
                                  java.lang.String data)
                           throws org.xml.sax.SAXException
```

This will indicate that a processing instruction (other than the XML declaration) has been encountered.

Specified by:

`processingInstruction` in interface `org.xml.sax.ContentHandler`

Parameters:

`target` - `String` target of PI

`data` - `String`**Throws:**

`SAXException` - when things go wrong

---

startPrefixMapping

```
public void startPrefixMapping(java.lang.String prefix,
                               java.lang.String uri)
```

This will indicate the beginning of an XML Namespace prefix mapping. Although this typically occur within the root element of an XML document, it can occur at any point within the document. Note that a prefix mapping on an element triggers this callback *before* the callback for the actual element itself (`startElement(java.lang.String, java.lang.String, java.lang.String, org.xml.sax.Attributes)`) occurs.

Specified by:

`startPrefixMapping` in interface `org.xml.sax.ContentHandler`

Parameters:

`prefix` - `String` prefix used for the namespace being reported

`uri` - `String` URI for the namespace being reported

Throws:

`SAXException` - when things go wrong

---

endPrefixMapping

```
public void endPrefixMapping(java.lang.String prefix)
```

This indicates the end of a prefix mapping, when the namespace reported in a `startPrefixMapping(java.lang.String, java.lang.String)` callback is no longer available.

Specified by:

`endPrefixMapping` in interface `org.xml.sax.ContentHandler`

Parameters:

`prefix` - `String` of namespace being reported

Throws:

`SAXException` - when things go wrong

---

startElement

```
public void startElement(java.lang.String namespaceURI,
                         java.lang.String localName,
                         java.lang.String rawName,
                         org.xml.sax.Attributes atts)
                  throws org.xml.sax.SAXException
```

This reports the occurrence of an actual element. It will include the element's attributes, with the exception of XML vocabulary specific attributes, such as `xmlns:[namespace prefix]` and `xsi:schemaLocation`.

Specified by:

`startElement` in interface `org.xml.sax.ContentHandler`

Parameters:

`namespaceURI` - `String` namespace URI this element is associated with, or an empty `String`

`localName` - `String` name of element (with no namespace prefix, if one is present)

`rawName` - `String` XML 1.0 version of element name: [namespace prefix]:[localName]

`atts` - `Attributes` list for this element

Throws:

`SAXException` - when things go wrong

---

endElement

```
public void endElement(java.lang.String namespaceURI,
                       java.lang.String localName,
                       java.lang.String rawName)
                throws org.xml.sax.SAXException
```

Indicates the end of an element (</[element name]>) is reached. Note that the parser does not distinguish between empty elements and non-empty elements, so this will occur uniformly.

Specified by:

endElement in interface org.xml.sax.ContentHandler

Parameters:

namespaceURI - String URI of namespace this element is associated with

localName - String name of element without prefix

rawName - String name of element in XML 1.0 form

Throws:

SAXException - when things go wrong

---

characters

```
public void characters(char[] ch,
                       int start,
                       int end)
                throws org.xml.sax.SAXException
```

This will report character data (within an element).

Specified by:

characters in interface org.xml.sax.ContentHandler

Parameters:

ch - char[] character array with character data

start - int index in array where data starts.

end - int index in array where data ends.

Throws:

SAXException - when things go wrong

---

ignorableWhitespace

```
public void ignorableWhitespace(char[] ch,
                                int start,
```

```
                                    int end)
                             throws org.xml.sax.SAXException
```

This will report whitespace that can be ignored in the originating document. This is typically only invoked when validation is ocurring in the parsing process.

Specified by:

`ignorableWhitespace` in interface `org.xml.sax.ContentHandler`

Parameters:

`ch` - `char[]` character array with character data

`start` - `int` index in array where data starts.

`end` - `int` index in array where data ends.

Throws:

`SAXException` - when things go wrong

---

skippedEntity

```
public void skippedEntity(java.lang.String name)
                   throws org.xml.sax.SAXException
```

This will report an entity that is skipped by the parser. This should only occur for non-validating parsers, and then is still implementation-dependent behavior.

Specified by:

`skippedEntity` in interface `org.xml.sax.ContentHandler`

Parameters:

`name` - `String` name of entity being skipped

Throws:

`SAXException` - when things go wrong

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class MyContentHandler

```
java.lang.Object
   |
   +--MyContentHandler
```

All Implemented Interfaces:

org.xml.sax.ContentHandler

class **MyContentHandler**

extends java.lang.Object

implements org.xml.sax.ContentHandler

`MyContentHandler` implements the SAX `ContentHandler` interface and defines callback behavior for the SAX callbacks associated with an XML document's content.

Constructor Summary

(package private)	**`MyContentHandler`**`()`

Method Summary

void	**`characters`**`(char[] ch, int start, int end)` This will report character data (within an element).
void	**`endDocument`**`()` This indicates the end of a Document parse - this occurs after all callbacks in all SAX Handlers..
void	**`endElement`**`(java.lang.String namespaceURI, java.lang.String localName, java.lang.String rawName)` Indicates the end of an element (`</[element name]>`) is reached.

void	**endPrefixMapping**(java.lang.String prefix) This indicates the end of a prefix mapping, when the namespace reported in a startPrefixMapping(java.lang.String, java.lang.String) callback is no longer available.
void	**ignorableWhitespace**(char[] ch, int start, int end) This will report whitespace that can be ignored in the originating document.
void	**processingInstruction**(java.lang.String target, java.lang.String data) This will indicate that a processing instruction (other than the XML declaration) has been encountered.
void	**setDocumentLocator**(org.xml.sax.Locator locator) Provide reference to Locator which provides information about where in a document callbacks occur.
void	**skippedEntity**(java.lang.String name) This will report an entity that is skipped by the parser.
void	**startDocument**() This indicates the start of a Document parse - this precedes all callbacks in all SAX Handlers with the sole exception of setDocumentLocator(org.xml.sax.Locator).
void	**startElement**(java.lang.String namespaceURI, java.lang.String localName, java.lang.String rawName, org.xml.sax.Attributes atts) This reports the occurrence of an actual element.
void	**startPrefixMapping**(java.lang.String prefix, java.lang.String uri) This will indicate the beginning of an XML Namespace prefix mapping.

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MyContentHandler

`MyContentHandler()`

Method Detail

setDocumentLocator

```
public void setDocumentLocator(org.xml.sax.Locator locator)
```

Provide reference to `Locator` which provides information about where in a document callbacks occur.

Specified by:

`setDocumentLocator` in interface `org.xml.sax.ContentHandler`

Parameters:

`locator` - `Locator` object tied to callback process

---

startDocument

```
public void startDocument()
                    throws org.xml.sax.SAXException
```

This indicates the start of a Document parse - this precedes all callbacks in all SAX Handlers with the sole exception of `setDocumentLocator(org.xml.sax.Locator)`.

Specified by:

`startDocument` in interface `org.xml.sax.ContentHandler`

Throws:

`SAXException` - when things go wrong

---

endDocument

```
public void endDocument()
                  throws org.xml.sax.SAXException
```

This indicates the end of a Document parse - this occurs after all callbacks in all SAX Handlers..

Specified by:

`endDocument` in interface `org.xml.sax.ContentHandler`

Throws:

`SAXException` - when things go wrong

---

processingInstruction

```
public void processingInstruction(java.lang.String target,
                                  java.lang.String data)
                           throws org.xml.sax.SAXException
```

This will indicate that a processing instruction (other than the XML declaration) has been encountered.

Specified by:

`processingInstruction` in interface `org.xml.sax.ContentHandler`

Parameters:

`target` - `String` target of PI

`data` - `String`**Throws:**

`SAXException` - when things go wrong

---

startPrefixMapping

```
public void startPrefixMapping(java.lang.String prefix,
                               java.lang.String uri)
```

This will indicate the beginning of an XML Namespace prefix mapping. Although this typically occur within the root element of an XML document, it can occur at any point within the document. Note that a prefix mapping on an element triggers this callback *before* the callback for the actual element itself (`startElement(java.lang.String, java.lang.String, java.lang.String, org.xml.sax.Attributes)`) occurs.

Specified by:

`startPrefixMapping` in interface `org.xml.sax.ContentHandler`

Parameters:

`prefix` - `String` prefix used for the namespace being reported

`uri` - `String` URI for the namespace being reported

Throws:

`SAXException` - when things go wrong

---

endPrefixMapping

```
public void endPrefixMapping(java.lang.String prefix)
```

This indicates the end of a prefix mapping, when the namespace reported in a `startPrefixMapping(java.lang.String, java.lang.String)` callback is no longer available.

Specified by:

`endPrefixMapping` in interface `org.xml.sax.ContentHandler`

Parameters:

`prefix` - `String` of namespace being reported

Throws:

`SAXException` - when things go wrong

---

startElement

```
public void startElement(java.lang.String namespaceURI,
                         java.lang.String localName,
                         java.lang.String rawName,
                         org.xml.sax.Attributes atts)
                  throws org.xml.sax.SAXException
```

This reports the occurrence of an actual element. It will include the element's attributes, with the exception of XML vocabulary specific attributes, such as `xmlns:[namespace prefix]` and `xsi:schemaLocation`.

Specified by:

`startElement` in interface `org.xml.sax.ContentHandler`

Parameters:

`namespaceURI` - `String` namespace URI this element is associated with, or an empty `String`

`localName` - `String` name of element (with no namespace prefix, if one is present)

`rawName` - `String` XML 1.0 version of element name: [namespace prefix]:[localName]

`atts` - `Attributes` list for this element

Throws:

`SAXException` - when things go wrong

---

endElement

```
public void endElement(java.lang.String namespaceURI,
                       java.lang.String localName,
                       java.lang.String rawName)
                throws org.xml.sax.SAXException
```

Indicates the end of an element (`</[element name]>`) is reached. Note that the parser does not distinguish between empty elements and non-empty elements, so this will occur uniformly.

Specified by:

`endElement` in interface `org.xml.sax.ContentHandler`

Parameters:

`namespaceURI` - `String` URI of namespace this element is associated with

`localName` - `String` name of element without prefix

`rawName` - `String` name of element in XML 1.0 form

Throws:

`SAXException` - when things go wrong

---

characters

```
public void characters(char[] ch,
                       int start,
                       int end)
                throws org.xml.sax.SAXException
```

This will report character data (within an element).

Specified by:

`characters` in interface `org.xml.sax.ContentHandler`

Parameters:

`ch` - `char[]` character array with character data

`start` - `int` index in array where data starts.

`end` - `int` index in array where data ends.

Throws:

`SAXException` - when things go wrong

---

ignorableWhitespace

```
public void ignorableWhitespace(char[] ch,
                                int start,
                                int end)
                         throws org.xml.sax.SAXException
```

This will report whitespace that can be ignored in the originating document. This is typically only invoked when validation is ocurring in the parsing process.

Specified by:

`ignorableWhitespace` in interface `org.xml.sax.ContentHandler`

Parameters:

`ch` - `char[]` character array with character data

`start` - `int` index in array where data starts.

`end` - `int` index in array where data ends.

Throws:

`SAXException` - when things go wrong

---

skippedEntity

```
public void skippedEntity(java.lang.String name)
                   throws org.xml.sax.SAXException
```

This will report an entity that is skipped by the parser. This should only occur for non-validating parsers, and then is still implementation-dependent behavior.

Specified by:

`skippedEntity` in interface `org.xml.sax.ContentHandler`

Parameters:

`name` - `String` name of entity being skipped

Throws:

`SAXException` - when things go wrong

Class MyErrorHandler

```
java.lang.Object
  |
  +--MyErrorHandler
```

All Implemented Interfaces:

org.xml.sax.ErrorHandler

class **MyErrorHandler**

extends java.lang.Object

implements org.xml.sax.ErrorHandler

`MyErrorHandler` implements the SAX `ErrorHandler` interface and defines callback behavior for the SAX callbacks associated with an XML document's errors.

Constructor Summary

(package private)	**`MyErrorHandler`**()

Method Summary

void	**`error`**(org.xml.sax.SAXParseException exception) This will report an error that has occurred; this indicates that a rule was broken, typically in validation, but that parsing can reasonably continue.
void	**`fatalError`**(org.xml.sax.SAXParseException exception) This will report a fatal error that has occurred; this indicates that a rule has been broken that makes continued parsing either impossible or an almost certain waste of time.
void	**`warning`**(org.xml.sax.SAXParseException exception) This will report a warning that has occurred; this indicates that while no XML rules were "broken", something appears to be incorrect or missing.

Methods inherited from class java.lang.Object

```
, clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Constructor Detail

MyErrorHandler

`MyErrorHandler``()`

Method Detail

warning

```
public void warning(org.xml.sax.SAXParseException exception)
             throws org.xml.sax.SAXException
```

This will report a warning that has occurred; this indicates that while no XML rules were "broken", something appears to be incorrect or missing.

Specified by:

`warning` in interface `org.xml.sax.ErrorHandler`

Parameters:

`exception` - `SAXParseException` that occurred.

Throws:

`SAXException` - when things go wrong

---

error

```
public void error(org.xml.sax.SAXParseException exception)
           throws org.xml.sax.SAXException
```

This will report an error that has occurred; this indicates that a rule was broken, typically in validation, but that parsing can reasonably continue.

Specified by:

`error` in interface `org.xml.sax.ErrorHandler`

Parameters:

`exception` - `SAXParseException` that occurred.

Throws:

`SAXException` - when things go wrong

---

fatalError

```
public void fatalError(org.xml.sax.SAXParseException exception)
                throws org.xml.sax.SAXException
```

This will report a fatal error that has occurred; this indicates that a rule has been broken that makes continued parsing either impossible or an almost certain waste of time.

Specified by:

`fatalError` in interface `org.xml.sax.ErrorHandler`

Parameters:

`exception` - `SAXParseException` that occurred.

Throws:

`SAXException` - when things go wrong

Class SAXParserDemo

```
java.lang.Object
  |
  +--SAXParserDemo
```

public class **SAXParserDemo**
extends java.lang.Object

`SAXParserDemo` will take an XML file and parse it using SAX, displaying the callbacks in the parsing lifecycle.

Constructor Summary

`SAXParserDemo()`

Method Summary

`static void`	`main(java.lang.String[] args)` This provides a command line entry point for this demo.
`void`	`performDemo(java.lang.String uri)` This parses the file, using registered SAX handlers, and output the events in the parsing process cycle.

Methods inherited from class java.lang.Object
`, clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

SAXParserDemo

`public` **`SAXParserDemo`**`()`

Method Detail

performDemo

`public void` **`performDemo`**`(java.lang.String uri)`

This parses the file, using registered SAX handlers, and output the events in the parsing process cycle.

Parameters:

`uri` - `String` URI of file to parse.

---

main

`public static void` **`main`**`(java.lang.String[] args)`

This provides a command line entry point for this demo.

Hierarchy For All Packages

Class Hierarchy

- class java.lang.Object
  - class **MyContentHandler** (implements org.xml.sax.ContentHandler)
  - class **MyErrorHandler** (implements org.xml.sax.ErrorHandler)
  - class **SAXParserDemo**

Deprecated API

Class **Tree** **Deprecated** **Index** **Help**

PREV NEXT **FRAMES** **NO FRAMES**

C

E

F

I

M

P

S

startDocument() - Method in class MyContentHandler

This indicates the start of a Document parse - this precedes all callbacks in all SAX Handlers with the sole exception of `MyContentHandler.setDocumentLocator(org.xml.sax.Locator)`.

startElement(String, String, String, Attributes) - Method in class MyContentHandler

This reports the occurrence of an actual element.

startPrefixMapping(String, String) - Method in class MyContentHandler

This will indicate the beginning of an XML Namespace prefix mapping.

W

warning(SAXParseException) - Method in class MyErrorHandler

This will report a warning that has occurred; this indicates that while no XML rules were "broken", something appears to be incorrect or missing.

C E F I M P S W

Class **Tree** **Deprecated** **Index** **Help**

PREV NEXT **FRAMES** **NO FRAMES**

How This API Document Is Organized

This API (Application Programming Interface) document has pages corresponding to the items in the navigation bar, described as follows.

Package

Each package has a page that contains a list of its classes and interfaces, with a summary for each. This page can contain four categories:

- Interfaces (italic)
- Classes
- Exceptions
- Errors

Class/Interface

Each class, interface, inner class and inner interface has its own separate page. Each of these pages has three sections consisting of a class/interface description, summary tables, and detailed member descriptions:

- Class inheritance diagram
- Direct Subclasses
- All Known Subinterfaces
- All Known Implementing Classes
- Class/interface declaration
- Class/interface description
- Inner Class Summary
- Field Summary
- Constructor Summary
- Method Summary
- Field Detail
- Constructor Detail
- Method Detail

Each summary entry contains the first sentence from the detailed description for that item. The summary entries are alphabetical, while the detailed descriptions are in the order they appear in the source code. This preserves the logical groupings established by the

programmer.

Tree (Class Hierarchy)

There is a Class Hierarchy page for all packages, plus a hierarchy for each package. Each hierarchy page contains a list of classes and a list of interfaces. The classes are organized by inheritance structure starting with `java.lang.Object`. The interfaces do not inherit from `java.lang.Object`.

- When viewing the Overview page, clicking on "Tree" displays the hierarchy for all packages.
- When viewing a particular package, class or interface page, clicking "Tree" displays the hierarchy for only that package.

Deprecated API

The Deprecated API page lists all of the API that have been deprecated. A deprecated API is not recommended for use, generally due to improvements, and a replacement API is usually given. Deprecated APIs may be removed in future implementations.

Index

The Index contains an alphabetic list of all classes, interfaces, constructors, methods, and fields.

Prev/Next

These links take you to the next or previous class, interface, package, or related page.

Frames/No Frames

These links show and hide the HTML frames. All pages are available with or without frames.

Serialized Form

Each serializable or externalizable class has a description of its serialization fields and methods. This information is of interest to re-implementors, not to developers using the API. While there is no link in the navigation bar, you can get to this information by going to any serialized class and clicking "Serialized Form" in the "See also" section of the class description.

This help file applies to API documentation generated using the standard doclet.

```
/*--

 Copyright (C) 2000 Brett McLaughlin. All rights reserved.

 Redistribution and use in source and binary forms, with or without modifica-
 tion, are permitted provided that the following conditions are met:

 1. Redistributions of source code must retain the above copyright notice,
    this list of conditions, and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright notice,
    this list of conditions, the disclaimer that follows these conditions,
    and/or other materials provided with the distribution.

 3. Products derived from this software may not be called "Java and XML", nor may
    "Java and XML" appear in their name, without prior written permission from
    Brett McLaughlin (brett@newInstance.com).

 THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES,
 INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS  FOR A PARTICULAR  PURPOSE ARE  DISCLAIMED.  IN NO  EVENT SHALL  THE
 JDOM PROJECT  OR ITS CONTRIBUTORS  BE LIABLE FOR  ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL,  EXEMPLARY, OR CONSEQUENTIAL  DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT  OF SUBSTITUTE GOODS OR SERVICES; LOSS
 OF USE, DATA, OR  PROFITS; OR BUSINESS  INTERRUPTION)  HOWEVER CAUSED AND ON
 ANY  THEORY OF LIABILITY,  WHETHER  IN CONTRACT,  STRICT LIABILITY,  OR TORT
 (INCLUDING  NEGLIGENCE OR  OTHERWISE) ARISING IN  ANY WAY OUT OF THE  USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

 This software was originally created by Brett McLaughlin <brett@newInstance.com>.
 For more  information on "Java and XML", please see
<http://www.oreilly.com/catalog/javaxml/>
 or <http://www.newInstance.com>.

 */
import java.io.IOException;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * <b><code>SAXParserDemo</code></b> will take an XML file and parse it using SAX,
 *   displaying the callbacks in the parsing lifecycle.
 *
 * @author Brett McLaughlin
 * @version 1.0
 */
public class SAXParserDemo {

    /**
     * <p>
     * This parses the file, using registered SAX handlers, and output
     *   the events in the parsing process cycle.
     * </p>
     *
```

```
     * @param uri <code>String</code> URI of file to parse.
     */
    public void performDemo(String uri) {
        System.out.println("Parsing XML File: " + uri + "\n\n");

        // Get instances of our handlers
        ContentHandler contentHandler = new MyContentHandler();
        ErrorHandler errorHandler = new MyErrorHandler();

        try {
            // Instantiate a parser
            XMLReader parser =
                XMLReaderFactory.createXMLReader(
                    "org.apache.xerces.parsers.SAXParser");

            // Register the content handler
            parser.setContentHandler(contentHandler);

            // Register the error handler
            parser.setErrorHandler(errorHandler);

            // Parse the document
            parser.parse(uri);

        } catch (IOException e) {
            System.out.println("Error reading URI: " + e.getMessage());
        } catch (SAXException e) {
            System.out.println("Error in parsing: " + e.getMessage());
        }


    }

    /**
     * <p>
     * This provides a command line entry point for this demo.
     * </p>
     */
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java SAXParserDemo [XML URI]");
            System.exit(0);
        }

        String uri = args[0];

        SAXParserDemo parserDemo = new SAXParserDemo();
        parserDemo.performDemo(uri);
    }

}

/**
 * <b><code>MyContentHandler</code></b> implements the SAX
 *   <code>ContentHandler</code> interface and defines callback
 *   behavior for the SAX callbacks associated with an XML
 *   document's content.
 */
class MyContentHandler implements ContentHandler {

    /** Hold onto the locator for location information */
```

```
  private Locator locator;

  /**
   * <p>
   * Provide reference to <code>Locator</code> which provides
   *   information about where in a document callbacks occur.
   * </p>
   *
   * @param locator <code>Locator</code> object tied to callback
   *                process
   */
  public void setDocumentLocator(Locator locator) {
      System.out.println("    * setDocumentLocator() called");
      // We save this for later use if desired.
      this.locator = locator;
  }

  /**
   * <p>
   * This indicates the start of a Document parse - this precedes
   *   all callbacks in all SAX Handlers with the sole exception
   *   of <code>{@link #setDocumentLocator}</code>.
   * </p>
   *
   * @throws <code>SAXException</code> when things go wrong
   */
  public void startDocument() throws SAXException {
      System.out.println("Parsing begins...");
  }

  /**
   * <p>
   * This indicates the end of a Document parse - this occurs after
   *   all callbacks in all SAX Handlers.</code>.
   * </p>
   *
   * @throws <code>SAXException</code> when things go wrong
   */
  public void endDocument() throws SAXException {
      System.out.println("...Parsing ends.");
  }

  /**
   * <p>
   * This will indicate that a processing instruction (other than
   *   the XML declaration) has been encountered.
   * </p>
   *
   * @param target <code>String</code> target of PI
   * @param data <code>String</code containing all data sent to the PI.
   *             This typically looks like one or more attribute value
   *             pairs.
   * @throws <code>SAXException</code> when things go wrong
   */
  public void processingInstruction(String target, String data)
      throws SAXException {

      System.out.println("PI: Target:" + target + " and Data:" + data);
  }

  /**
```

```
     * <p>
     * This will indicate the beginning of an XML Namespace prefix
     *   mapping.  Although this typically occur within the root element
     *   of an XML document, it can occur at any point within the
     *   document.  Note that a prefix mapping on an element triggers
     *   this callback <i>before</i> the callback for the actual element
     *   itself (<code>{@link #startElement}</code>) occurs.
     * </p>
     *
     * @param prefix <code>String</code> prefix used for the namespace
     *               being reported
     * @param uri <code>String</code> URI for the namespace
     *            being reported
     * @throws <code>SAXException</code> when things go wrong
     */
    public void startPrefixMapping(String prefix, String uri) {
        System.out.println("Mapping starts for prefix " + prefix +
                           " mapped to URI " + uri);
    }

    /**
     * <p>
     * This indicates the end of a prefix mapping, when the namespace
     *   reported in a <code>{@link #startPrefixMapping}</code> callback
     *   is no longer available.
     * </p>
     *
     * @param prefix <code>String</code> of namespace being reported
     * @throws <code>SAXException</code> when things go wrong
     */
    public void endPrefixMapping(String prefix) {
        System.out.println("Mapping ends for prefix " + prefix);
    }

    /**
     * <p>
     * This reports the occurrence of an actual element.  It will include
     *   the element's attributes, with the exception of XML vocabulary
     *   specific attributes, such as
     *   <code>xmlns:[namespace prefix]</code> and
     *   <code>xsi:schemaLocation</code>.
     * </p>
     *
     * @param namespaceURI <code>String</code> namespace URI this element
     *                     is associated with, or an empty
     *                     <code>String</code>
     * @param localName <code>String</code> name of element (with no
     *                  namespace prefix, if one is present)
     * @param rawName <code>String</code> XML 1.0 version of element name:
     *                [namespace prefix]:[localName]
     * @param atts <code>Attributes</code> list for this element
     * @throws <code>SAXException</code> when things go wrong
     */
    public void startElement(String namespaceURI, String localName,
                             String rawName, Attributes atts)
        throws SAXException {

        System.out.print("startElement: " + localName);
        if (!namespaceURI.equals("")) {
            System.out.println(" in namespace " + namespaceURI +
                               " (" + rawName + ")");
```

```
        } else {
            System.out.println(" has no associated namespace");
        }

        for (int i=0; i<atts.getLength(); i++)
            System.out.println("  Attribute: " + atts.getLocalName(i) +
                               "=" + atts.getValue(i));
    }

    /**
     * <p>
     * Indicates the end of an element
     *   (<code>&lt;/[element name]&gt;</code>) is reached.  Note that
     *   the parser does not distinguish between empty
     *   elements and non-empty elements, so this will occur uniformly.
     * </p>
     *
     * @param namespaceURI <code>String</code> URI of namespace this
     *                     element is associated with
     * @param localName <code>String</code> name of element without prefix
     * @param rawName <code>String</code> name of element in XML 1.0 form
     * @throws <code>SAXException</code> when things go wrong
     */
    public void endElement(String namespaceURI, String localName,
                           String rawName)
        throws SAXException {

        System.out.println("endElement: " + localName + "\n");
    }

    /**
     * <p>
     * This will report character data (within an element).
     * </p>
     *
     * @param ch <code>char[]</code> character array with character data
     * @param start <code>int</code> index in array where data starts.
     * @param end <code>int</code> index in array where data ends.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void characters(char[] ch, int start, int end)
        throws SAXException {

        String s = new String(ch, start, end);
        System.out.println("characters: " + s);
    }

        /**
     * <p>
     * This will report whitespace that can be ignored in the
     *   originating document.  This is typically only invoked when
     *   validation is ocurring in the parsing process.
     * </p>
     *
     * @param ch <code>char[]</code> character array with character data
     * @param start <code>int</code> index in array where data starts.
     * @param end <code>int</code> index in array where data ends.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void ignorableWhitespace(char[] ch, int start, int end)
        throws SAXException {
```

```
        String s = new String(ch, start, end);
        System.out.println("ignorableWhitespace: [" + s + "]");
    }

        /**
     * <p>
     * This will report an entity that is skipped by the parser.  This
     *   should only occur for non-validating parsers, and then is still
     *   implementation-dependent behavior.
     * </p>
     *
     * @param name <code>String</code> name of entity being skipped
     * @throws <code>SAXException</code> when things go wrong
     */
    public void skippedEntity(String name) throws SAXException {
        System.out.println("Skipping entity " + name);
    }

}

/**
 * <b><code>MyErrorHandler</code></b> implements the SAX
 *   <code>ErrorHandler</code> interface and defines callback
 *   behavior for the SAX callbacks associated with an XML
 *   document's errors.
 */
class MyErrorHandler implements ErrorHandler  {

    /**
     * <p>
     * This will report a warning that has occurred; this indicates
     *   that while no XML rules were "broken", something appears
     *   to be incorrect or missing.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void warning(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Warning**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Warning encountered");
    }

    /**
     * <p>
     * This will report an error that has occurred; this indicates
     *   that a rule was broken, typically in validation, but that
     *   parsing can reasonably continue.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
```

```
     */
    public void error(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Error**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Error encountered");
    }

    /**
     * <p>
     * This will report a fatal error that has occurred; this indicates
     *   that a rule has been broken that makes continued parsing either
     *   impossible or an almost certain waste of time.
     * </p>
     *
     * @param exception <code>SAXParseException</code> that occurred.
     * @throws <code>SAXException</code> when things go wrong
     */
    public void fatalError(SAXParseException exception)
        throws SAXException {

        System.out.println("**Parsing Fatal Error**\n" +
                           "  Line:    " +
                              exception.getLineNumber() + "\n" +
                           "  URI:     " +
                              exception.getSystemId() + "\n" +
                           "  Message: " +
                              exception.getMessage());
        throw new SAXException("Fatal Error encountered");
    }

}
```

```
<?xml version="1.0"?>
<!-- test xml page -->
<!DOCTYPE averagegpa SYSTEM "averagegpa.dtd">
<averagegpa>
        <student>
                <firstname> Mary </firstname>
                <lastname> Wong </lastname>
                <sn> 97123456 </sn>
                <gpa> 3.78 </gpa>
                <grade Grade="A" />
        </student>
        <student>
                <firstname> Brian William</firstname>
                <lastname> Mulroney </lastname>
                <sn> 579874562 </sn>
                <gpa> 2.02</gpa>
                <grade Grade="C-" />
        </student>
        <student>
                <lastname> Bjork </lastname>
                <sn> 763245610 </sn>
                <gpa> 2.78 </gpa>
                <grade Grade="B-" />
        </student>
        <student>
                <firstname>John </firstname>
                <lastname>von Neumann </lastname>
                <sn> 26387782 </sn>
                <gpa>4.2</gpa>
                <grade Grade="A+" />
        </student>
</averagegpa>
```

```
package xml.sax.gpaExample2;

import java.io.IOException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * A SAX parser example with a simple calculation.<p>
 * The file to be parsed conforms to the DTD averagegpa.dtd. An example is
 * averagegpa.xml. <p>
 * The program finds the student with the highest gpa. Output is to stdout.<p>
 *
 * SAX parsers analyse the xml document dynamically, producing events for each type
 * of XML data. The analysis proceeds in depth first order. The program must arrange
 * to capture relevant data as it "flies by". <p>
 *
 * Various interfaces provide callback methods to respond to these events. SAX 2.0
provides
 * the convenience class DefaultHandler which provides empty implementations off all
 * the methods in these interfaces. You subclass this class and override the methods of
 * your choosing to provide the desired functionality. <p>
 *
 * This program uses several of the most useful of these callbacks, characters(),
startElement(),
 * and endElement().<p>
 */
public class GPAExample {

        private final String xmlURL =
"file:/c:/coursesf2000/cps720/xml/sax/gpaExample/averagegpa.xml";

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java GPAExample [XML URI]");
            System.exit(0);
        }

        String uri = args[0];

       GPAExample gpaAnalysis = new GPAExample();
       gpaAnalysis.analyse(uri);
    }

        /**
         * Sets up the parser. These calls are very standardized.
         * @param uri <code> String</code>The locator for the XML file to be parsed.
         */
        public void analyse(String uri) {

                  // Get instances of our handlers. DefaultHandler is a convenience
                  // class which implements default empty methods for4 interfaces,
                  // EntityResolver, DTDHandler, ContentHandler and ErrorHandler.
                  // Subclass and override the methods you need.

                DefaultHandler theHandler = new MyHandler(); // creae the subclass

                  try {
```

```
                                // Instantiate a parser
                                XMLReader parser =
                                        XMLReaderFactory.createXMLReader(
                                                "org.apache.xerces.parsers.SAXParser");

                                // Register the content handler (part of DefaultHandler)
                                parser.setContentHandler(theHandler);

                                // Register the error handler (Should be done sometime.)
                                //parser.setErrorHandler(errorHandler);

                                // Parse the document
                                parser.parse(uri);

                        } catch (IOException e) {
                                System.out.println("Error reading URI: " +
e.getMessage());
                        } catch (SAXException e) {
                                System.out.println("Error in parsing: " +
e.getMessage());
        }
        }

}
/**
 * Tailors the DefaultHandler for this application.
 */
class MyHandler extends DefaultHandler {

                /*
                 * The startElement() and endElement() methods are called every time the
parser
                 * sees an Element. These variables, global to the class, allow the
different types
                 * of elements to be distinguished across calls. In particular the
control the action of
                 * the characters() method which reads PCDATA from the XML file.
                 */
                private boolean isGPA = false;
                private boolean isFirstName = false;
                private boolean isLastName = false;

                // retain some of the parsed data when the SAX parser moves on

                private Student bestStudent, aStudent;

                public MyHandler() {
                        bestStudent = new Student("","", 0.00f, "");
                }

           public void startDocument() throws SAXException {
                System.out.println("Parsing begins...");
            }

                public void endDocument() throws SAXException {
                        // Called at the end, so print out the result.
                        System.out.println("The Best Student");
                        System.out.println("================");
                        bestStudent.printStudent();
                        System.out.println("...Parsing ends.");
                }
```

```
           // This captures the PCDATA etc in an element.

                public void characters(char[] ch, int start, int end)
                                throws SAXException {

                                String s = new String(ch, start, end);

                                // keep the parsed data. The aStudent variable should not
be null because
                                // it got the reference when startElement() was called on
<student> tag.
                                // Similarly the boolean flags are set in starElement().

                                if(isFirstName) {
                                        if(aStudent != null) {
                                                aStudent.setFirstName(s);
                                        }
                                } else if(isLastName) {
                                        if(aStudent != null) {
                                                aStudent.setLastName(s);
                                        }
                                } else if(isGPA) {
                                        if(aStudent != null) {
                                                float gp = (new Float(s)).floatValue();
                                                aStudent.setGpa(gp);
                                        }
                                }
                  }

                    /**
                       * <p>
                       * This reports the occurrence of an actual element.  It will
include
                       *   the element's attributes, with the exception of XML vocabulary
                       *   specific attributes, such as
                       *   <code>xmlns:[namespace prefix]</code> and
                       *   <code>xsi:schemaLocation</code>.
                       * </p>
                       *
                       * @param namespaceURI <code>String</code> namespace URI this
element
                       *                     is associated with, or an empty
                       *                     <code>String</code>
                       * @param localName <code>String</code> name of element (with no
                       *                   namespace prefix, if one is present)
                       * @param rawName <code>String</code> XML 1.0 version of element
name:
                       *                 [namespace prefix]:[localName]
                       * @param atts <code>Attributes</code> list for this element
                       * @throws <code>SAXException</code> when things go wrong
                       */
                      public void startElement(String namespaceURI, String localName,
                                               String rawName, Attributes atts)
                          throws SAXException {

                          if(localName.equals("student")) {
                                           aStudent = new Student("", "", 0.0f,
"");
                                    }
```

```
                                        // Flags to control the characters() method.
                                        // Distinguish among types of elements.

                                        if(localName.equals("gpa")) {
                                                isGPA = true;
                                        } else if(localName.equals("firstname")) {
                                                isFirstName = true;
                                        } else if(localName.equals("lastname")) {
                                                isLastName = true;
                                        }
                        }

                        public void endElement(String namespaceURI, String localName,
                                               String rawName)
                            throws SAXException {

                                // Check that the event signals the end of a  <student>
element. If so, the
                                // current student's (aStudent) data fields are
populated. So now you can
                                // compare the current student to the best so far
student and perhaps replace
                                // the best so far student.

                            if(localName.equals("student") && aStudent != null) {
                                        if(aStudent.getGpa() > bestStudent.getGpa()) {
                                                bestStudent = aStudent;
                                        }
                                        aStudent.printStudent();

                                        // allow garbage collection of old student data
                                        aStudent = null;
                                }
                                // At any time, two of these are already false, but
it's a waste of time to test.
                            isGPA = false;
                            isFirstName = false;
                            isLastName = false;
                        }
                /**
                 * A class to represent student data. The fields correspond to those in the XML
document.
                 */
                class Student  {
                        private String firstName;
                        private String lastName;
                        private float gpa;
                        private String grade;

                        public Student(String fn, String ln, float gpa, String g) {
                                firstName = fn;
                                lastName = ln;
                                this.gpa = gpa;
                                grade = g;
                        }
                        public void setFirstName(String fn) {
                                firstName = fn;
                        }
                        public void setLastName(String ln) {
                                lastName = ln;
                        }
```

```
                public void setGpa(float g) {
                        gpa = g;
                }
                public float getGpa() {
                        return gpa;
                }
                public void setGrade(String g) {
                        grade = g;
                }
                public void printStudent() {
                        System.out.println(firstName);
                        System.out.println(lastName);
                        System.out.println(gpa);
                        System.out.println(grade);
                }
        }
}
```

A Few Notes on XSL and XSLT

XSL (Extensible Styesheet Language) is an important part of XML. XSLT (Extensible Stylesheet Language Transformer) is a processing program which uses the XLS to transform an XML document into different forms, for example HTML or PDF (Adobe Acrobat) format.

Actually, in principle XSL is more powerful that this. It could be used to transform an XML document into almost any format.

XSL is quite a bit more extensive and complicated that XML itself. These notes only provide a very limited introduction. The notes discuss two very simple examples, and show how to use the XSLT processor, Xalan, process an XML document and a corresponding XSL document.

The Xalan distribution is available from the Apache XML Project. You also need the Xerces parser (comes with the Xalan distribution, or separately).

Local versions: xerces.jar xalan.jar

The Basic Mechanism of XSL

First of all, note that an XSL document is an XML document. The XSL document has a tree structure of the original XML document that it is being used to process (by the XSLT). The XSLT must find each node in the XML document (usually an Element) and its attributes, if any, and then apply rules defining how to present each node. The XSLT finds these rules in the XSL document. Often these rules transform the XML document into an HTML document for presentation to a human user. The XSLT processor processes both trees (in the XML and XSL documents) in a depth first order.

The XSLT is a pattern matcher. It uses a template method to match and find the nodes it wishes to process. XSL is really the programming language for the XSLT processor. It is a kind of scripting language. Its commands start with xsl: followed by the command name and various arguments. Some of these are illustrated in the examples.

A Simple Example

We return to our old friend, averagegpa.xml (with its DTD averagegpa.dtd). Although averagegpa.xml is reasonably readable by a human, it would be nicer if we could turn it into a regular HTML document, averagegpa,html, for viewing with a browser. The XSL program, averagegpa.xsl.

averagegpa.xsl

```
<?xml version="1.0"?>

<!--
A style sheet for averagegpa.xml and averagegpa.dtd.
Used with Apache Xalan XSLT.
```

```
The output file is averagegpa.html.
DG. Nov. 2000
-->

<!--
xls is an international namespace.
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

    <xsl:template match="averagegpa">
        <html>
            <head>
                <title> Student Average GPAs </title>
            </head>
            <body>
                <h1> Student Average Marks</h1>
                <xsl:apply-templates select="student" />
            </body>
        </html>
    </xsl:template>
    <xsl:template match="student">
        <h3>
            <xsl:value-of select="firstname"/>
            <xsl:value-of select= "lastname"/>
        </h3>
        <hr />
        <ul>
            <li>Student Number: <xsl:value-of select="sn"/></li>
            <li>Average GPA: <xsl:value-of select="gpa"/></li>
            <!--
            The next line illustrates XPath.
            -->
            <li> Grade: <xsl:value-of select="./grade/@Grade"/></li>
        </ul>
        <xsl:apply-templates select="grade"/>
    </xsl:template>
```

```
<!--
Another way to deal with the grade Element and its attribute.
Disadvantage is that it makes a separate ul.
-->
<xsl:template match="grade">
    <ul>
        <li> Average Grade: <xsl:value-of select="@Grade"/> </li>
    </ul>
</xsl:template>
</xsl:stylesheet>
```

Notes on the program

This example shows a number of basic points about XSL programming.

- xsl is a namespace which must be unique. Namespaces must refer to their "home base" URL. The line **<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">** accomplishes this for the xsl namespace.
- Part of programming in XSL consists of making up templates to match against nodes in the XML document. **<xsl:template match=**"averagegpa"> is a simple example. The matching can be much more complicated involving wildcards and logical functions such as not.
- After they are created, templates must be applied to target nodes. A simple example is **<xsl:apply-templates select="student"** />. This template is only applied to student nodes among the children of <averagegpa>. Since all the children of <averagegpa> are <student>nodes you really do not need to sepecify. Note that the xsl:apply-template only applies to children, not grandchildren or more remote sub-nodes. In this example, for instance, the nodes <firstname> <lastname> etc which are childern of <student> are not affected.
- XSL comes with a large number of commands (functions). One of the most used is xsl:value-of, for example, **<xsl:value-of select="firstname"/>**. There are many others. They allow for normal programming logic: sequences, branches, and loops. Examples are xsl:for-each, xsl:counter, xsl:if, xsl:choose, xsl:when, xsl:otherwise, See XML Pocket Reference, p.56-70. Using them has the flavour of shell programming with XML syntax.
- XSL makes use of a separate feature, called XPath. XPath helps you locate nodes at different levels in the XML tree. It is modelled on the path structure of UNIX . **<xsl:value-of select="./grade/@Grade"/>** is an example. This isnvokded at the <student> level so we go down one level to the node <grade> from the current <student> level denoted by the '.' as in the UNIX file system. Interstingly, to get at the Grade attribute of the <grade> node we go down one more level and then use the '@' prefix which is used to distinguish attribute names from element names.
- The example program shows another way of accessing the <grade> element's Grade attribute: use a template. **<xsl:apply-templates select="grade"/>**. The select is important. You have to watch out for multiple passes throught the same nodes.

Running the example

You can use Xalan from the command line to create the HTML file. (Xalan has many more features for integrating it with Java code but these are not covered here.)

You need to download the Xalan package and unzip it. I put it in C:\xalan-j_2_0_D01\. The two necessary files are xalan.jar and xerces.jar in the bin subdirectory. These must be on the classpath, for example, set classpath=C:\xalan-j_2_0_D01\bin\xerces.jar;C:\xalan-j_2_0_D01\bin\xalan.jar.

Then, in the directory where all the xml, dtd and xsl files are, type,

java org.apache.xalan.xslt.Process -IN averagegpa.xml -XSL averagegpa.xsl >averagegpa.html

Note the fully qualified name for the Process class.

A Second Example

This example shows some more complicated things. It is taken from Brett McLaughlin's *Java and XML*. It provides an XSL script for table of contents example discussed previously.

contents.xml JavaXML.dtd

JavaXML.html.xsl

This example illustrates using some logical constructs in XSL. The result of using this XSL script is shown in contents.html.

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:JavaXML="http://www.oreilly.com/catalog/javaxml/"
version="1.0"
>

    <xsl:template match="JavaXML:Book">
        <html>
            <head>
                <title><xsl:value-of select="JavaXML:Title" /></title>
            </head>
            <body>
                <xsl:apply-templates select="*[not(self::JavaXML:Title)]" />
            </body>
        </html>
    </xsl:template>
```

```
<xsl:template match="JavaXML:Contents">
    <center>
        <h2>Table of Contents</h2>
    </center>
    <hr />
    <ul>
        <xsl:for-each select="JavaXML:Chapter">
            <xsl:choose>
                <xsl:when test="@focus='Java'">
                    <li><xsl:value-of select="JavaXML:Heading" /> (Java
                    Focus)</li>
                </xsl:when>
                <xsl:otherwise>
                    <li><xsl:value-of select="JavaXML:Heading" /> (XML
                    Focus)</li>
                </xsl:otherwise>
                </xsl:choose>
        </xsl:for-each>
    </ul>
</xsl:template>

<xsl:template match="JavaXML:References">
    <p>
    <center><h3>Useful References</h3></center>
    <ol>
        <xsl:for-each select="JavaXML:Reference">
            <li>
                <xsl:element name="a">
                    <xsl:attribute name="href">
                        <xsl:value-of select="JavaXML:Url" />
                    </xsl:attribute>
                    <xsl:value-of select="JavaXML:Name" />
                </xsl:element>
            </li>
        </xsl:for-each>
    </ol>
```

```
            </p>
        </xsl:template>

        <xsl:template match="JavaXML:Copyright">
            <xsl:copy-of select="*" />
        </xsl:template>

    </xsl:stylesheet>
```

Note: [CPS720 FALL 2000]. Students are only responsible for constructs in the first example. The second example is shown only for interest.

Student Average Marks

Mary Wong

- Student Number: 97123456
- Average GPA: 4.01
- Grade: A+
- Average Grade: A+

Brian William Mulroney

- Student Number: 579874562
- Average GPA: 2.02
- Grade: C-
- Average Grade: C-

Kennedy

- Student Number: 763245610
- Average GPA: 2.78
- Grade: B-
- Average Grade: B-

```
<?xml version="1.0"?>

<!--
        A style sheet for averagegpa.xml and averagegpa.dtd.
        Used with Apache Xalan XSLT.
        The output file is averagegpa.html.
        DG. Nov. 2000
-->

<!--
        xls is an international namespace.
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

 <xsl:template match="averagegpa">
        <html>
        <head>
        <title> Student Average GPAs </title>
        </head>
        <body>
                <h1> Student Average Marks</h1>
                <xsl:apply-templates select="student" />
        </body>
        </html>
 </xsl:template>
 <xsl:template match="student">
        <h3>
                <xsl:value-of select="firstname"/>
                <xsl:value-of select= "lastname"/>
        </h3>
        <hr />
        <ul>
                <li>Student Number: <xsl:value-of select="sn"/></li>
                <li>Average GPA: <xsl:value-of select="gpa"/></li>
                <!--
                        The next line illustrates XPath.
                -->
                <li> Grade: <xsl:value-of select="./grade/@Grade"/></li>
        </ul>
        <xsl:apply-templates select="grade"/>
 </xsl:template>

 <!--
        Another way to deal with the grade Element and its attribute.
        Disadvantage is that it makes a separate ul.
 -->
 <xsl:template match="grade">
 <ul>
        <li> Average Grade: <xsl:value-of select="@Grade"/> </li>
 </ul>
 </xsl:template>
</xsl:stylesheet>
```

Table of Contents

Useful References

1.

http://www.oreilly.com/catalog/javaxml/ grammar not found

Table of Contents

Useful References

1. The W3C
2. XSL List

Agent Negotiations

Bargaining and negotiation play key roles in natural agent interactions. People negotiate with other people all the time. Negotiation is a method by which autonomous agents seek to cooperate by coordinating their activities and goals.

Virtual software agents will no doubt have to learn to negotiate with one another. In the InfoSleuth architecture, for example, there are 'clouds' of agents, consumer agents, producer agents, and broker agents. Negotiations must occur throughout such systems.

Honesty and Deception

Honesty is the best policy -- not necessarily.

In the case of a multi-agent system set up by one company, a situation in which all agents are designed by the same people, or where the agents have little autonomy, the honesty problem does not really occur. Presumably, the system is designed to be cooperative and no one has any incentive to deceive.

But, in the future, one can imagine large numbers of software agents with considerable autonomy, designed by people who do not know one another, and, presumably, designed to serve the selfish ends of the agent's owner. There is no reason not to expect that agents which practice deception will appear.

Optimal Agreements

It also turns out, that protecting against deception can lead to inefficient, and even downright bad in some situations. The threat of deception poisons the atmosphere of negotiations. A famous example of this is the Prisoner's Dilemma game.

One way to solve the deception problem is policing. People who get caught are jailed or fined. This is a very expensive approach, and may not be very effective.

In some situations it is possible to design *institutional mechanisms, or protocols*, which make honesty the best policy. There are many examples of this, especially in connection with taxation policies, and research continues on such mechanisms.

Negotiation protocols which encourage honesty would be very useful in a world of autonomous virtual agents.

Making Honesty the Best Policy

The mechanisms which can make human bargaining more honest could also be useful as protocols in negotiations among virtual agents. Here are two examples of protocols from the human world designed to make honesty the best policy

Example 1. Jumping landing queue at airports.

Consider the following scenario.

One factor in deciding which aircraft lands first is the amount of fuel remaining in the approaching aircraft. Assume that aircraft with lower amounts of fuel are moved up in the queue and therefore land sooner.

There are benefits to landing earlier. First of all, money is saved because the aircraft does not waste fuel while circling the airport waiting to land. Secondly, on time performance is enhanced, making customers happy. These are obvious incentives to cheat.

Now suppose, the fuel situation is "discussed" between two software agents, one in the plane, the other in the tower. Further suppose that the airline has programmed its plane's software agent to underestimate its fuel remaining. Choosing how much to underestimate would be tricky. If the underestimation were too small, it would do no good; too large, and the agent would get caught. Perhaps a clever agent program would take into account factors such as the weather: bad weather, underestimate by a greater among.

Preventing cheating

The simplest way is the law. Airlines caught doing this get fined. Note that a software implementation of this cheating algorithm is less likely to get caught than a purely human one. That is because far fewer people would have to know about it, perhaps only the programmers and some executives. Without the software agent, all the pilots would have to be part of the plot.

A better way to prevent this scam would be some kind of automatic mechanism to encourage honesty. One way would be a surcharged based upon how many places the plane was moved up in the queue. The surcharge would provide a disincentive to underreporting the remaining fuel supply to gain a queue advantage.

Example 2. Vickrey's Auction

Many bids for large contracts are organized using a sealed bid mechanism (protocol). Bids kept secret (sealed) and submitted at some deadline. The bids are opened and the lowest bidder gets the contract.

A bidder would like to bid as high as possible without being underbid by a rival. So would the rival bidders. The temptation is to spend resources investigating (spying on?) rivals to try to guess how low they can bid, and then bit slightly lower still. These investigations waste resources with the result that in the long run, the average lowest bids will be higher than they could be. There is also an upward pressure

on the bids, as all the bidders have a tendency to keep their bids high to make a good profit if they do with the bid. This is a case of inefficiency more than dishonesty.

William Vickrey (a Nobel Prize winner in economics) came up with the following mechanism which is claimed to be more efficient in the long run. Bids are sealed as before, and opened simultaneously. The lowest bid is chosen, but, the *lowest bidder is paid the value of the second lowest bid* !! The winning bidder does not get the amount of money determined by itself. Rather the reward is determined by another bidder's lowest bid.

The claim is that incentives to investigate opponents and push bids up is removed by this protocol.

These two examples illustrate possible mechanisms or protocols which align self interest with honesty and optimality. Thee mechanisms encourage effective cooperation rather than costly conflict. Achieving optimal cooperative solutions to possibly conflictual situations is the subject of *Game Theory*.

A Brief Introduction to Game Theory

Game Theory is the study of confilct and cooperation among agents. The theory was invented and first developed by none other that John von Neumann. Johnny (as he was known when he lived in the USA) was a party animal in addition to being the world's smartest man. His parties were legendary. At these parlour games were often played. Poker was also popular. In additon to partying and developing theories of computer science, quantum mechanics and economics, Johnny enjoyed playing games with children (e.g., scissors, paper, stone - see below).

So it was not surprising that at some point he became interested in a theory of games. However, what started as an analysis of party and childrens' games became deadly serious during the Cold War. The USA and the Soviet union were playing a deadly nuclear "game". Game Theorist were called upon to provide advise on how to conduct this game rationally. Perhaps they contributed to the survival of the human race during this dangerous period.

Conflict, Cooperation and Negotiation among Agents

With the rise of the Internet the idea of software agents representing their human masters has become a real possibility. Your agents some stage will have to engage my agents in automated negotiations at some stage. Confilcts of interest will arise. So will the need to cooperate. These questions are just what Game Theory deals with. It is not surprising that Game Theory has become of interest in Computer Science. (In fact, the Draft Curriculum 2001 of the ACM/IEEE lists it as a topic.)

These notes provide an informal introduction to Game Theory using a few examples.

Two Person Zero Sum Games

The phrase "zero sum game" has become part of the English language. It refers to a situation of conflict in which two or more agents compete for the same prize. One person's win is another's loss. Two person zero sum games are the easiest to formalize mathematically so most introductions to Game Theory start with these kinds of games.

A (partial) game theory ontology

Consider the following.

7	2	5	1
2	2	3	4
5	3	4	4
3	2	1	6

what is it?

Most people would interpret this as a table of numbers. Some might call it a matrix of integers.

Let's add some labels.

Column Player Strategy

Row Player Strategy

	A	B	C	D
I	7	2	5	1
II	2	2	3	4
III	5	3	4	4
IV	3	2	1	6

Now this table takes on a certain meaning. The matrix is called a ***payoff matrix***. There are two players, that is, agents, which are assumed to be rational. Players have **strategies**. This is a special use of the word strategy. Normally, a strategy is a plan. In Game Theory, strategy must be complete. In other words a player's strategy must del with every possible stategy of the opponent.

The outcome of the interaction of the players' strategies are the payoffs shown as number in the matrix. For eacple, if the Row player plays strategy I and the column player plays his strategy, 'B', the payoff is 2.

Zero Sum

By convention, the payoffs shown in the matrix are from the point of view of the Row player. When the Row player gets a payoff of 2, for example, the implication is that the Column player got a payoff of -2. The sum of the two payoffs is 0, hence the name Zero Sum Game.

The game shown would be very unfair to the Column player if the payoffs were in dollars. A game with dollar payoffs would presumably have some negative entries in the payoff matrix. Negative entries would mean that the Row player would pay the Column player.

How the games are played

- **Simultaneous moves**. In the basic version of game theory, both players move at the same time. They do not take turns. (Game theory can be extended to include taking move turns.
- **Iterative and single games.** Games can be played only once but they are much more interesting if they are played over and over. Games played many times are called iterative games. It is also often useful to have both players ignorant of just how many times the game will be repeated.
- **Perfect information**. Both players can see the whole payoff matrix. This means that they each know the strategies of their opponents, as well as their own.

Connection to the Real World

Game Theory itself is an ontology which is part of mathematics. The mention of dollars for the payoff is an attempt to link this mathematics to real objects in the human world. Traditionally, Game Theorists have linked their subject to economics and political science. i.e. business and politics.

Interpreting Game Theory results in the real world is very difficult and often controversial. Game Theory abstracts from much of the detail of the real world but still can offer insights if carefully done. Hopefully, it can also be the basis of conflict resolution among software agetns.

To use Game Theory you have to figure out all the possible strategies. Then you have to estimate payoffs. These need not be absolute. (In the above example, if all the numbers were 10 times larger the solutions would be the same.) Once the game is setup, the question arises: ***what is the best strategy for each player?***

A Harmless Interprestaion of the example

(Taken from the book, The Compleat Strategyst, by J.D. Williams.)

The strategies represent roads through a mountain range. Four roads go east-west (Rows), and four go north-south (Columns). The two groups of roads intersect in the 16 places of the matrix and the payoffs are altitudes of the intersections.

The two players want to meet and camp at one of the intersections. The catch is that the Row player (the east-west driver) wants to camp at the highest possible altitude, whereas the Column player (the north-south driver) wishes to camp at the lowest possible altitude!

They cannot communicate with each other. So what will happen? One possiblity is that each could choose a route at random. The result would be meeing at some random altitude. (This is a rather fanciful example since it assumes that the two always can arrive at any intersection at the same time :-)) The random approach will probably result in unhappiness for one of the two players, too high for one, or too low for the other. Game Theory analysis provides a better answer.

Minimax and maximin

They plaryers are rational and *respectful*. They each assume that the other will not make a mistake. So each assumes the other will always make the 'best' move.

The players are also assumed to be conservative in the sense that they will not give up a possibly modest sure thing to gamble for a much larger gain at the risk of "loosing their shirt".

The Row player (the maximizer) chooses a strategy which has the biggest worst case, the largerst loweset value among the strategies. The Row player chooses the strategy with the maximum minimum! (i.e., the maximin.)

The Column player (the minimizer) chooses the strategy which has the smallest worst case, the smallest highest value among the strategies. The Column player chooses the strategy with the minimum maximum (i.e., the minimax).

What happesn in William's example?

Row Player's Viewpoint

Strategy	I	II	III	IV
Min Payoff	1	2	3	1

Maximin = 3 strategy III

Colum Player's Viewpoint

Strategy	A	B	C	D
Max Payoff	7	3	5	6

Minimax = 3 strategy B

So the Row player choosees strategy III. The Column player chooses strategy B,

In this example, the minimax of the Column player equals the maximin of the Row player. The game has a unique solution. The two players wind up at the 3 (thousand foot) junction and no player can improve on this outcome.

When the maximin and the minimax are equal the game is said to have a "**saddle point**", a term used in calculus and horseback riding.

Non Deterministic games.

Unfortunately, a game with a saddle point is a lucky break for a Game Theoretic analysis. Most games are like the following obtained by changing one payoff in the original game matrix.

Column Player Strategy

Row Player Strategy

	A	B	C	D
I	7	2	5	1
II	2	2	3	4
III	5	3	4	4
IV	3	**6**	1	6

Row Player Viewpoint

Strategy	I	II	III	IV
Min Payoff	1	2	3	1

Maximin = 3 strategy III

Column Player Viewpoint

Strategy	A	B	C	D
Max Payoff	7	6	5	6

Minimax = 5 strategy

Nothing changes for the Row player, but the Column player now chooses strategy C instead of B and the minimax is 5. If the Column player plays strategy C, she winds up at payoff 4 (thousand feet). Could she do better?

There is a "gray area" in this new version of the game separating the minmax from the maxmin (the payoffs between 3 and 5). How should players deal with this. The answer is ***mixed stategies***.

Mixed Strategies

If a game has a saddle point, then each player has an optimal pure strategy.That is, one strategy is certainly the best according to the rules of rationality (minimaxing). With no pure strategy available a player must create a mix of two or more of the available strategies.

Mixing introduces probablilty. You imagine that the game is to be played over and over again. You can, of course, only play one stategy at a time. But you can changed the chosen strategy at each play of the game. The question is, *how often should you play each strategy? And in what order?*

What you do is to assign a probablilty to each strategy and then, each time the game is played, you choose a strategy accoring to the probablility. The advantage of this randomization is that you keep the opponent guessing. Now the question becomes, how do you rationally choose the appropriate probabilities?

Finding the optimal mixed strategy, which means finding these probablilities is called solving the game.

One way of doing this uses the Simplex method of Linear Programming. (You may have taken this in an Operations Research course, e.g., MTH 503). Solving these games is beyond the scope of CPS720. However, there is a program on the net, courtesy Stephan Waner and Steven Constenoble of Hofstra University in New York State.

Game Theory Simulation from Hofstra University

Uising the simulator with our example, we get the following mixed strategy.

The Optimal Row Strategy

[0, 0, 0.8333, 0.1667]

The Optimal Column Strategy

[0, 0.5, 0.5, 0]

This means that the Row player should play her strategy III 5 games out of every 6, strategy IV once every 6 games, and never play the other two.

The Column player could toss a coin and play each of strategies B and C 50% of the time each. She never plays strategies A and D.

The value of the game is 3.5 (thousand feet) which is the average payoff if the game is played over and over again. Notice that this is better for the Column player thant the value 4 (thousand) feet she obtained by using the minimax pure strategy mentioned earlier. The Row player can do nothing to prevent this improvement int Column's score.

Dominance

The payoff matrix shows an interesting feature. Row's strategy II is dominated by strategy III. Every score for III is greater than or equal to that of II. So you could simply cross out row II without changing the outcome.

Some nice notes from Hoftra University.

Scissors, paper and stone

This classic childrens' game was analysed by von Neumann in his original discussion of Game theory.

Rules of the game

1. Scissor cuts paper.
2. Paper covers stone.
3. Stone sharpens scissors.

- Two people simultaneously call out one of these three names. The winnder is the first name listed in the 3 rules.
- If the same name is called by both players, the game is a draw.

Scissors, Paper and stone is playes as an iterative game. It is also a fair game. The value of this game is therefore, 0.

A possible game matrix is:

Column Player Strategy

Row Player Strategy	scissors	paper	stone
scissors	0	1	-1
paper	-1	0	1
stone	1	-1	0

There is no pure strategy for either player in this game. The maximin for the row player is -1 for all three strategies. Similarly the minmax for the column player is 1 for all three strategies.

Because of the game's simplicity and symmetry it is easy to see that each player's mixed strategy should be to play each of his or her strategies with a probablility of 1/3. For example, each could *toss a die*. If 1 or 2 turned up, say scissors. If 3 or 4 turned up, say paper, if 5 or 6 turned up, say stone.

Notice that you must completely randomize your choices and the order of choices. Otherwise the opponent could spot a pattern and exploit it over a long period of plays.

Other Kinds of Games

n-person zero sum games

These can be quite complex because of the possiblity of coallitions and alliances. Those of you who watched *Survivor* will remember Rich and his friends. These games are hard to analyse.

Cooperative Games

Another interesting class of games. The most famous example of this type is the "Prisoners' Dilemma" discussed on another page of these notes. In these games the size of the payoffs can depend on the level of cooperation among players, in contrast to the zero sum nature of the games discussed here.

Cooperative Games

Zero sum games involve conflict. Your gain is my loss. Note that the payoff matrix in two player zero sum games needs only hold the payoff for one side (the Row Player, by convention). The payoff for the Column Player is just the negaive of that of the Row Player. But you can have games where this is not true. In such situations it is possible for the players to cooperate for the benefit of all.

How to achieve this cooperative benefit can turn out to be quite difficult if you assume that rational players pursue only their self interest. In other words, can you have cooperation without altruism. The dilemma of achieving cooperation based on selfishness is famously illustrated by the Prisoners' Dilemma Game.

Two Person Cooperative Games

In these games there are four kinds of payoffs.

- temptation
- reward (for cooperating)
- sucker
- punishment

(These names make most sense with the Prisoners' Dilemma version of cooperative games.)

The Payoff Matrix

Player 2

Player 1	cooperate	defect
cooperate	(3,3)	(0,5)
defect	(5,0)	(1,1)

Note in contrast to the zero sum payoff matrix, each element of this matrix has two values, one for each player. The row player (Player 1) payoff is listed first.

In the example shown, we have, for each player, the following payoffs,

- tempataion = 5
- reward = 3
- sucker = 0
- punishment = 1

The ordering temptation > reward > punishment > sucker makes this a Prisoners' Dilemma.

Strategies

Players have two possible strategies,

- cooperate
- defect

Single and Iterated Games

There are two ways the game can be played. You can have a one encounter game, or you can have an iterated game, that is, the game is played over and over and a score is kept. The outcome of these two ways of running the game can be quite different.

The Prisoners' Dilemma Game

The interest in this game is the problem of how to get the cooperative solution which benefits both players (and notice that the total score world be 6 in the above example, so the "collective" of the two players winds up with more "wealth" than any other solution gives (only 5 for the tempation payoff).

But no one wants to be suckered. Notice that if both defect, they get the "punishment" payoff, a rather impoverished result. But it's better than nothing, the sucker's payoff. When both players defect, the "cut their noses off to spite their faces". Everyone loses.

The trouble is, double defections seem to be the norm. Here are a few examples, from both art and reality.

Examples of the Prisoners' Dilemma Game

The Prisoner's Dilemma

The Prisoner's Dilemma Story

Two friends are arrested, charged with a crime, and held in separate cells so they cannot communicate with each other. They are interrogated separately. They face this situation. The police do not have enough real evidence, so they are looking for a confession. So they offer, to each prisoner separately, a "plea bargain". If one prisoner confesses and "fingers" the other, he will get off with a 1 year sentence, and his (former) friend will serve 5 years.

On the other hand, if neither prisoner "talks", the police will not have enough evidence, and both prisoners will get off "scott free", and serve 0 time. (If both confess, the judge reduces the 5 year sentence for good behaviour. -- not really necessary for the game.)

These options can be represented in a tabular form. (In this matrix smaller valuesare more desireable.)

		Player 2	
		Cooperates	Defects
Player 1	Cooperates	(0, 0)	(5, 1)
	Defects	(1, 5)	(4, 4)

The numbers are called payoffs. In this case the players want to minimize the payoff. When one player defects, the other is said get the sucker payoff.

The Prisoner's Dilemma game illustrates both the benefits and the difficulties in achieving cooperation. To achieve the optimum solution, both for yourself, and for both players together (a kind of "societal" payoff) both players must trust the other. But trust involves the risk of being suckered.

In practice, what usually happens is that both players defect, and get a very bad result for each of them, in the example, 4 years in jail each.

The Iterated Prisoner's Dilemma

In Prisoner's Dilemma games played only once, defection is almost inevitable, unless some outside, institutional mechanism has been invented to encourage cooperation. However, there is some hope if the game can be repeated over and over. To see this, consider restaurants.

Suppose there are two kinds of restaurants. The first group is in the tourist area. Customers are likely only to come once and never return. In this case the restaurant is tempted to "defect" by serving over priced meals which are not so good. This is a one shot Prisoner's Dilemma between the customer and the restauranteur in which the customer is suckered.

The second group of restaurants services an area with a large number of houses, apartments and condos.

The clientele live around the restaurants. If they like a restaurant, they will return again and again. If they feel they are suckered, they won't return. So these restaurants are likely to give better value to their customers.

A real Prisoner's Dilemma

The Cold War and the Nuclear Arms Race (1949-1989) can be viewed as a Prisoner's Dilemma. Fear of being suckered torpedoed any serious nuclear disarmament agreements. The optimal solution would have been for both sides to disarm saving billions of dollars and getting rid of nuclear risks. What was achieved was instead a vast waste of human resources, the arms race.

Prisoner's Dilemma in Art - Tosca's Kiss

One of the world's most famous operas is Tosca, by Giacomo Puccini. Its plot nicely illustrates the Prisoner's Dilemma. The opera takes place during the time of Napoleon. There are 3 main characters, Tosca, Cavaradossi, an artist and revolutionary in love with Tosca, and Scarpia, chief of police, and also in love with Tosca. Tosca is in love with Cavaradossi, and hates Scarpia whom she sees as an ugly lecher.

Scarpia catches Cavaradossi and plans to execute him by firing squad. He also opens "negotiations" with Tosca. If she will make love to him, he will free Cavaradossi. She accepts his offer. There is a catch. For appearance sake, the execution must go ahead, but Scarpia will order that the bullets of the firing squad will be blanks. Cavaradossi will have to pretend to be dead, but afterwards, he and Tosca can escape.

Tosca comes to Scarpia's chambers. He embraces her, and she stabs him to death a stiletto. This episode has become famous as "Tosca's kiss". At this moment the firing squad is heard firing.

Tosca runs to fetch her lover. He is lying on the ground. Of course, he really is dead. Scarpia had defected too. The firing squad used live bullets. In despair, Tosca leaps to her death off the battlements of the prison.

The Prisoner's Dilemma game can occur whenever negotiations take place, whether among human or artificial agents. Game Theory also discusses many other situations where negotiations and bargaining occur. It is a theory of rational behaviour among autonomous agents. It assumes that cooperative agreements are "out there". The question is how can rational agents get "there" and avoid the likes of Tosca's kiss.

Autonomous artificial agents will encounter the same dilemmas. Protocols to enable cooperation will have to be developed. Undoubtedly, Game Theory will be extensively used in the development of such protocols.

You too can play the Prisoners' Dilemma

Some Notes on Ascape

Introduction

Ascape is a Java API for simulating multi-agent systems on a single computer. It is particularly suitable for simulations in the social sciences. The origins of Ascape lie in the Artificial Life world, in particular, the Swarm research project.

Ascape was developed by Miles Parker of the Brookings Institute in Washington, DC.

The Ascape web site

Ascape comes with API documentation and a number of examples with source code. These are valuable but rather too complex for people starting out with Ascape. At least I found them so. One fo the mostt approachable of these examples is the Demographic Prisoner's Dilemma. Recently Miles Parker published a paper based on this example which provides a tutorial-like description.

Demographic Prisoner's Dilemma

This example is still pretty complex for beginners. So in the hope of filling the gap caused by the absence of more elementary examples, here are three tutorials.

- Tutorial 1. part 1. The Voter Game
- Tutorial 1 part 2. The Voter Game with data gathering
- Tutorial 2. Population Explosion

It is hoped that these three tutorials will help newcomers get up to speed with Ascape. They are based on my own experiences trying to learn Ascape.

Any suggestions or corrections would be greatly appreciated.

The Notes

Part 1 of Tutorial 1 implements the Voter Game simply with an overhead view of the agents. In Part 2 the vote count of each party is recorded and displayed in a histogram.

Basic Ascape Architecture

Ascape Rules

The Voter Game

Tutorial 1. Part 1.

Ascape views

Tutorial 1. Part 2

Tutorial 2

Some notes on the Ascape Architecture

The Ascape program is a very sophiticated construction with an elegeant object oriented architecture. Its author, Miles Parker, has explained Ascape's architecture on a number of occasions. See, for example,

The Chicago Presentation

The JASSS paper

These comments reflect my own learnig experience based on reading the above and trying to program the Voter Game. The aim is to highlight what you need to know to get started.

Scapes and Agents

Scapes are collections of agents. But they are also agents themselves which is rather confusing (but very powerful). It is interesing the look at the inheritance sturcture of agents and scapes (in the package edu.brook.ascape.model). Here is a principal route:

Object
AscapeObject
Agent
Cell
CellOccupant
Scape
ScapeGraph
ScapeVector

Most movels start with a root Scape which is usually a subclass of the class ScapeVector. This root scape ties together the various parts of the model and holds assorted global variables needed by the model.

Typically this root ScapeVector holds at least two objects, a ScapeGraph, and another ScapeVector. This will be the case in our simple Voter Game model. This second ScapeVector holds the agents, 2500 of them in the Voter Game.

The ScapeGraph holds the "playing field" or environment for the agents. In many cases (such as the famous SugarScape example, and the Voter example) this consists of a toroidal lattice. Ascape provides quite a few environment geometries all derived from ScapeGraph. For example,

ScapeGraph
ScapeArray2D
ScapeArray2DMoore

Note that, given this inheritance structure, a ScapeArray2DMoore object could be a CellOccupant -- one could have a scape of many toroidal worlds!)

Agents and Cells

Individual agents like in environment such as ScapeArray2D objects. But you can't just plunk them into a lattice location without preparation.

To enable cells in a Scape to hold agents you must fill them with HostCell objects. One way to do this is to call the Scape method setPrototypeAgent(Agent a) on a ScapeGraph lattice with the argument a HostCell object, like this:

```
ScapeGraph lattice = new ScapeArray2DMoore();

lattice.setPrototypeAgent(new HostCell());
```

The ScapeVector which holds the agents also needs some preparation. For example suppose we have,

```
Voter avoter = new Voter();
```

where Voter is a subclass of CellOccupant (which is a subclass of Agent). Then you could have

```
ScapeVector voters = new ScapeVector();

voters.setPrototypeAgent(avoter);
```

clones the Voter agent and puts them in the voters ScapeVector for future use.

Summary

An Ascape model has root ScapeVector holding at least 2 things, a ScapeGraph representing the agents' environment and a ScapeVector containing the actual agents (which will be used for things like collecting data). The ScapeGraph is often some kind of lattice structure. Cells of this lattice must be made "comfortable" in this lattice using the HostCell class.

Your actual agent is a subclass of CellOccupant.

Ascape Rules

Ascape agent behaviour is defined by rules. Ascape itself defines a number of useful rules. These rules are implemented in the Agent class, in the CellOccupant class, and in the Scape class. The rules have names which are static fields.

Rules in the Agent class

- DEATH_RULE
- FISSIONING_RULE
- FORCE_DIE_RULE
- FORCE_FISSION_RULE
- FORCE_MOVE_RULE
- INITIALIZE_RULE
- ITERATE_AND_UPDATE_RULE
- ITERATE_RULE
- METABOLIZE_RULE
- MOVEMENT_RULE
- UPDATE_RULE

See the discussion of these fields in the documentation.

Rules in the CellOccupant class

- MOVE_RANDOM_LOCATION_RULE
- PLAY_HOST_RULE
- PLAY_NEIGHBORS_RULE
- RANDOM_WALK_AVAILABLE_RULE
- RANDOM_WALK_RULE

Rules in the Scape class

- COLLECT_STATS_RULE
- CREATE_RULE
- CREATE_SCAPE_RULE
- INTERNAL_START_RULE
- INTERNAL_SCAPE_RULE
- PAUSE_RULE

- RESUME_RULE
- START_RULE
- STOP_RULE

Since Scapes are Agents, they inherit all the Agent rules.

By the way, the Scape and Agent classes carry extensive introductoriy documentation which you will likely find quite useful.

It is also possible to create your own rules and this is often done. In the Voter Game example only two predefined Ascape rules are needed.

Using Rules

Registration

Rules belong to Scapes . You tell a Scape about a rule (register the rule) using the metthods addRule() or addInitialRule(). These methods have various forms and belong to the Scape class.

You can add rules when defining scapes but it is often more convenient ot add the rules from inside the code for the individual agents. To this end, there is the Agent class method, scapeCreated() in which you can call the addRule() methods via getScape(). The latter returns the scape which the agent in question inhabits. For example, (inside the agent's class definition, Voter.java)

```
public void scapeCreated() {
        // ...
        getScape().addRule(ITERATE_AND_UPDATE_RULE);
        // ...
}
```

Rule Actions

Once the scape knows the rules, it applies them iteratively to every agent on the scape (in an unsystematic order). It does this by invoking certain methods of the Scape or Agent classes which correspond to the appropriate rule.

As programmer, you override these methods to add your own code to carry out the rule's action. Or, you can accept the default behaviour provided by Ascape.

Some rules automatically call a boolean method which tests some condition for the rule's action. These rules behave like recognize (some condition or situation)-action rules as found, for example, in the JESS expert system shell.

Bug alert! Make sure you type the signatures of these special methods exactly. Otherwise, your rule may do nothing or execute the default behaviour only.

Example

In the case of the ITERATE_AND_UPDATE_RULE above, Ascape calls two methods, **iterate()** and **update()**.

Rule Order

There are two orders in calling rules, RULE_ORDER and AGENT_ORDER. The default is AGENT_ORDER. In RULE_ORDER all the rules are executed on one agent and then all the rules are executed on the next agent and so on. In AGENT_ORDER, the fisrt rule is executed on all the agents, then the second rule is executed on all the agents, and so on.

The ITERATE_AND_UPDATE_RULE should be used with RULE_ORDER. In this case, the iterate() method is called on all the agents, and the results temporarily stored. Then the update() method is called on all the agents to update the agents' states. This ordering simulates parallelism. For example, in the Voter Game this ordering ensures that each agent uses its neighbours old values in its voting decision, avoiding a chain reaction where some of its neighbours have already changed their opinions before they are used.

The Voter Game

The Game

I first came across this game in A. K. Dewdney's fun book, The Armchair Universe (Freeman, 1988). It is the fourth "easy piece" starting on p.221. Dewdney credits the idea to Peter Donnelly and Dominic Welsh.

The idea is simple. A wraped around grid world (toroid) is completely occupied with stationary agents, one in each cell of the grid. Each agent has 8 neighbours (the so-called Moore neighbourhood).

In the example given by Dewdney there are two parties. Initially all the agents are assigned a voiting preference randomly. Then the system is run through many cycles. At each cycle, each agent-voter, consults one of its (his,her?) neighbours, chosen randomly, and, being of weak mind, immediately agrees to vote the same way as that neighbour.

That is all there is too it! The question is, what voting patterns, if any, emerge from this simple behaviour? You might be surprised. Or, maybe not. Or maybe you will have to decide whether the result is a figment of the program's imagination, or has some broader significance.

In this first tutorial we implement this simulation with one modification. The user can set the number of parties between 2 and 5.

Ascape Tutorial: The Voter Game' part 1.

Rules of the Voter Game

Notes on the class structure of Ascape

The program consists of two files, VoterScape.java and Voter.java (in package dave.ascape.voters2).

The code below is colour coded. Important methods and classes are bolded when they first appear.

- red. a method which will be called by Ascape
- dark blue. often used methods from the Ascape API.
- green. often used Ascape classes

The Agents' Environment

VoterScape.java (source)

```
package dave.ascape.voters2;
import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;
/**
* A simple version of Dewdney's voting game.
* Voters ask a random neighbour his choice of party and switch to
* that party on the next round of voting.
* see also Voter.java for the agent code.
*/
public class VoterScape extends ScapeVector {
    // VoterScape is the root scape
    private int numberOfParties = 3;
    private int latticeWidth = 50;
    private int latticeHeight = 50;
    ScapeGraph lattice; // the agents' environment
    ScapeVector voters; // the agents
    Overhead2DView overheadView;
    public void createScape() {
```

```
            super.createScape(); // needed for basic setup
            lattice = new ScapeArray2DMoore();
            lattice.setPrototypeAgent(new HostCell());
            lattice.setExtent(new Coordinate2DDiscrete(latticeWidth, latticeHeight));
            Voter voter = new Voter(); // see Voter.java
            voter.setHostScape(lattice); // where agents live
            voters = new ScapeVector();
            voters.setPrototypeAgent(voter);
            voters.setExecutionOrder(Scape.RULE_ORDER);
            addAgent(lattice); // add the lattice to the root scape
            addAgent(voters); // add the voters list to the root scape
        }
        public void onSetup() {
            ((ScapeVector) voters).setExtent(new
            Coordinate1DDiscrete(latticeWidth*latticeHeight));
        }
        public void createViews() {
            super.createViews();
            //Now, add a simple overhead view so that we can view the lattice:
            overheadView = new Overhead2DView();
            //Set its cell size to 12
            overheadView.setCellSize(12);
            //add it to the lattice so that it can view and be controlled by it
            lattice.addView(overheadView);
        }
        public void setNumberOfParties(int n) {
            numberOfParties = n;
        }
        public int getNumberOfParties() {
            return numberOfParties;
        }
    }
```

The Agent Itself

The agent itself is in the file Voter.java (in package dave.ascape.voters2).

Notes on Ascape rules

Voter.java (source)

```
package dave.ascape.voters2;
import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;
import java.awt.Color;
/**
* An agent for the VoterScape world. Votes according to the opinion of
* a randomly chosen 'moore' neighbour.
*
* See also dave.ascape.voters2.VoterScape.java
*/
public class Voter extends CellOccupant {
        private int supportedParty = 0;
        private int temp = 0;
        private int numParties = 0;
        public Voter() {}
        public void initialize() {
                super.initialize();
                numParties = ((VoterScape)scape.getModel()).getNumberOfParties();
                supportedParty = randomInRange(0, numParties-1);
        }
        public void scapeCreated() {
                getScape().addInitialRule(MOVE_RANDOM_LOCATION_RULE);
                getScape().addRule(ITERATE_AND_UPDATE_RULE);
        }
        public void iterate() {
                CellOccupant [] neighbors = getHostCell().getNeighboringOccupants();
                int chosenNeighbor = randomInRange(0, neighbors.length-1);
                CellOccupant chosenAgent = neighbors[chosenNeighbor];
                temp = ((Voter) chosenAgent).getSupportedParty();
        }
        public void update() {
                supportedParty = temp;
        }
```

```
        public Color getColor() {
                switch(supportedParty) {
                        case 0: return Color.blue;
                        case 1: return Color.red;
                        case 2: return Color.black;
                        case 3: return Color.orange;
                        case 4: return Color.white;
                        default: return Color.green;
                }
        }
        public void setSupportedParty(int p) {
                supportedParty = p;
        }
        public int getSupportedParty() {
                return supportedParty;
        }
        public int getNumParties() {
                return numParties;
        }
        public void setNumParties(int n) {
                numParties = n;
        }
}
```

Notes on Voter.java

1. scape.getModel().getNumberOfParties().

This line has some interesting features. The number of parties is held by the numberOfParties variable in the root Scape (VoterScape). So the idea is to tell the agent how many parties there are. This way of doing this (rather than, for example, passing the information as a constructor argument) allows the user to change the number of parties at run time.

For the Voter agent to get at the getNumberOfParties() method of the VoterScape class requires the two steps shown: 'scape' and 'getModel()'. What is going on?

scape is protected field of th class, AscapeObject. Since the AscapeObject class is the root of the Ascape system, this fields is inherited by all Ascape objects. It represents the scape (agent) to which the agent in whose code the 'scape' variable appears, belongs.

In the VoterScape example, consider these two lines in the CreateScape() method,

```
voters = new ScapeVector();
voters.setPrototypeAgent(voter);
```

The ScapeVector, voters, "ownes" the voter agent. So, 'scape' refers to this Scape.

Another line of the code is,

```
addAgent(voters);
```

The votes Scape is added to the root Scape, VoterScape by this instruction.

This is where getModel() comes in. getModel() is a method of the Agent class. It gets the root scape, in this case the Scape owning the voters Scape. This is just the VoterScape class which contains the variables and methods which describe the global properties of the model being designed,

```
package dave.ascape.voters2;

import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;

/**
 * A simple version of Dewdney's voting game.
 * Voters ask a random neighbour his choice of party and switch to
 * that party on the next round of voting.
 * One run: 4 parties reduced to 1 in 12000 cycles
 *
 * This is mean to be an elementary introduction to Ascape programming.
 * No statistics are collected. Only an overhead view is shown.
 * Users can, however, adjust the number of parties from 2 to 5 at run time
 * in the usuaal Ascape way.
 *
 * see also Voter.java for the agent code.
 * Statistics are added in the program dave.ascape.voters3 package.
 */
public class VoterScape extends ScapeVector {

    private int numberOfParties = 3;
    private int latticeWidth = 50;
    private int latticeHeight = 50;

    ScapeGraph lattice;
    ScapeVector voters;

    Overhead2DView overheadView;
    ChartView chart;

    public void createScape() {
        super.createScape();

        lattice = new ScapeArray2DMoore();
        lattice.setPrototypeAgent(new HostCell());
        lattice.setExtent(new Coordinate2DDiscrete(latticeWidth, latticeHeight));
        Voter voter = new Voter();
        voter.setHostScape(lattice);

        voters = new ScapeVector();
        voters.setPrototypeAgent(voter);

        // how does it know number of voters?
        // default rule order is AGENT_ORDER (SEE paragraph 5.7)
        // RULE_ORDER is needed  to simulate parallism with ITERATE_AND_UPDATE_RULE
        voters.setExecutionOrder(Scape.RULE_ORDER);
        addAgent(lattice);
        addAgent(voters);
    }

    public void onSetup() {
        //Set the extent of the scape, which is simply the number of agents we want.
        ((ScapeVector) voters).setExtent(new
Coordinate1DDiscrete(latticeWidth*latticeHeight));
    }


    public void createViews() {
```

```
        super.createViews();
       //Now, add a simple overhead view so that we can view the lattice:
        overheadView = new Overhead2DView();
        //Set its cell size to 12
        overheadView.setCellSize(12);
        //add it to the lattice so that it can view and be controlled by it
        lattice.addView(overheadView);
    }

    public void setNumberOfParties(int n) {
        numberOfParties = n;
    }
    public int getNumberOfParties() {
        return numberOfParties;
    }


}
```

```
package dave.ascape.voters2;

import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;

import java.awt.Color;

/**
 * An agent for the VoterScape world. Votes according to the opinion of
 * a randomly chosen 'moore' neighbour.
 *
 * See also dave.ascape.voters2.VoterScape.java
 */
public class Voter extends CellOccupant {

        private int supportedParty = 0;
        private int temp = 0;
        private int numParties = 0;

        public Voter() {}

        // This is not a good way -- implies hardcoding of number of parties
        // in the root VoterScape scape. When the number of parties was passed
        // this way, the value could not be changed at runtime. See the first
        // line of initialize for the correct way to get variable values
        // from the root Scape to individual agents.
        public Voter(int numParties) {
                this.numParties = numParties;
        }

        public void initialize() {
                super.initialize();
                // scape is a built in field in ScapeObject class.
                // getModel() is equivalent to getRootScape. The root scape is
                // where global variables are stored, such as the number of parties.
                // in this example. Belongs to the Agent class. Returns a Scape
                // so casting is necessary to the specific model root.
                numParties = ((VoterScape)scape.getModel()).getNumberOfParties();
                supportedParty = randomInRange(0, numParties-1);
        }

        public void scapeCreated() {
                getScape().addInitialRule(MOVE_RANDOM_LOCATION_RULE);

                getScape().addRule(ITERATE_AND_UPDATE_RULE);
        }

        // called by the ITERATE_AND_UPDATE_RULE

        // iterate is called on every agent. Then update() is called if the
ITERATE_AND_UPDAE
        // rule is used and RULE_ORDER is set.  Simulates p;arallelism.

        public void iterate() {
                CellOccupant [] neighbors = getHostCell().getNeighboringOccupants();
                // choose one at random. chekc itsopinion then change
                // could also have 3 choices with diffrent probabilities (adjustable)
                // no chnage, change, random change
                int chosenNeighbor = randomInRange(0, neighbors.length-1);
```

```
                CellOccupant chosenAgent = neighbors[chosenNeighbor];
                temp = ((Voter) chosenAgent).getSupportedParty();
        }

        public void update() {
                supportedParty = temp;
        }


        public Color getColor() {
                switch(supportedParty) {
                        case 0: return Color.blue;
                        case 1: return Color.red;
                        case 2: return Color.black;
                        case 3: return Color.orange;
                        case 4: return Color.white;
                        default: return Color.green;
                }
        }

        public void setSupportedParty(int p) {
                supportedParty = p;
        }
        public int getSupportedParty() {
                return supportedParty;
        }
        public int getNumParties() {
                return numParties;
        }
        public void setNumParties(int n) {
                numParties = n;
        }
}
```

Ascape Views and Stats

Ascape provides many ways of viewing the results of agent behaviour. There is the basic overhead view of the agent's world (often a toroid). As well, various statistics on the agents can be collected and viewed as time series, bar charts and pie charts. The tutorials associated with these notes show some simple uses of views.

The Overhead View

Most Ascape models implement this view which provides a dynamic picture of the agents. The first tutorial on the Voter Game only implements this view. To do so is quite simple. You simply override the Scape classes createViews() method like so:

```
public void createViews() {
        super.createViews();
        overheadView = new Overhead2DView();
        overheadView.setCellSize(12);
        lattice.addView(overheadView);
}
```

The lattice mentioned here is an instance of a ScapeArray2DMoore class. This createViews() method is part of the definition of the VoterScape model root ScapeVector class. Ascape calls createViews() automatically.

Collecting and Displaying Statistics

To display time series, bar charts, and pie charts you must first have Ascape collect approprate statistics. A simple illustration is provided in Tutorial 2.

Ascape Tutorial: The Voter Game part 2.

Voter Game with histogram view.

of the Voter Game

Notes on the class structure of Ascape

The program consists of two files, VoterScape.java and Voter.java (in package dave.ascape.**voters3**).

The code below is colour coded. Important methods and classes are bolded when they first appear.

- red. a method which will be called by Ascape
- dark blue. often used methods or fields from the Ascape API.
- green. often used Ascape classes

This version of the Voter Game adds a histogram view of the vote counts for the parties. The program also pauses on startup.

The code in smaller font is identical to the code in part 1 of this tutorial. Note that the file names are the same as those of part 1 but theyare in diffent packages.

The Agents' Environment

VoterScape.java (source)

```
package dave.ascape.voters2;
import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;
/**
* A simple version of Dewdney's voting game.
* Voters ask a random neighbour his choice of party and switch to
* that party on the next round of voting.
* One run: 4 parties reduced to 1 in 12000 cycles
*
* In this version, vote counts for the parties are collected and
* displayed (by default) in a histogram.
*
*/
public class VoterScape extends ScapeVector {
        // VoterScape is the root scape
        private int numberOfParties = 3;
        private int latticeWidth = 50;
        private int latticeHeight = 50;
        ScapeGraph lattice; // the agents' environment
```

```
ScapeVector voters; // the agents
Overhead2DView overheadView;
public void createScape() {
        super.createScape(); // needed for basic setup
        lattice = new ScapeArray2DMoore();
        lattice.setPrototypeAgent(new HostCell());
        lattice.setExtent(new Coordinate2DDiscrete(latticeWidth, latticeHeight));
        Voter voter = new Voter(); // see Voter.java
        voter.setHostScape(lattice); // where agents live
        voters = new ScapeVector();
        voters.setPrototypeAgent(voter);
        voters.setExecutionOrder(Scape.RULE_ORDER);
        addAgent(lattice); // add the lattice to the root scape
        addAgent(voters); // add the voters list to the root scape
}
public void onSetup() {
        ((ScapeVector) voters).setExtent(new Coordinate1DDiscrete(latticeWidth*latticeHeight));
}
/*
* Putting Ascape into pause right away allows the user to see the
* initial situation. The alternative is to cale setStartOnOpen(false) , for example in onSetup().
*/
public void onStart() {
        pause();
}
public void createViews() {
        super.createViews();
        //Now, add a simple overhead view so that we can view the lattice:
        overheadView = new Overhead2DView();
        //Set its cell size to 12
        overheadView.setCellSize(12);
        //add it to the lattice so that it can view and be controlled by it
        lattice.addView(overheadView);
        /*
        The following code shows how to add some statistic collection to the Voter Agent and have Ascape display it for your. The stats are collected using inner classes. Serveral ways of doing this are shown.
        The code is straightforward except for one trick which was ot obvious to me.
        The trick.
        */

        final StatCollector [] stats = {
```

```
                new StatCollectorCondCSA ("Party A") {
                        public boolean meetsCondition(Object obj) {
                                return ( ((Voter)obj).getSupportedParty() == 0);
                        }
                },
                new StatCollectorCondCSA ("Party B") {
                        public boolean meetsCondition(Object obj) {
                                return ( ((Voter)obj).getSupportedParty() == 1);
                        }
                },
                new StatCollectorCondCSA ("Party C") {
                        public boolean meetsCondition(Object obj) {
                                return ( ((Voter)obj).getSupportedParty() == 2);
                        }
                },
                new PartyD_Standing("Party D"),
                new PartyE_Standing("Party E")
        }; // end StatCollector array

        voters.addStatCollectors(stats);

        chart = new ChartView(ChartView.HISTOGRAM);
        voters.addView(chart);

        chart.addSeries("Count Party A", Color.blue);
        chart.addSeries("Count Party B", Color.red);
        chart.addSeries("Count Party C", Color.black);
        chart.addSeries("Count Party D", Color.orange);
        chart.addSeries("Count Party E", Color.yellow);
} // end createViews()
public void setNumberOfParties(int n) {
        numberOfParties = n;
}
public int getNumberOfParties() {
        return numberOfParties;
}
class PartyD_Standing extends StatCollectorCondCSA {

        public PartyD_Standing (String name) {
```

```
                super(name);
        }

        public boolean meetsCondition(Object obj) {
                return ( ((Voter)obj).getSupportedParty() == 3);
        }

    }

    class PartyE_Standing extends StatCollectorCondCSA {

        public PartyE_Standing(String name) {
                this.name = name;
        }
        public String getName() {
                return name;
        }
        public boolean meetsCondition(Object obj) {
                return ( ((Voter)obj).getSupportedParty() == 4);
        }

    }
} // end VoterScape
```

The Agent Itself

The agent itself is in the file Voter.java (in package dave.ascape.voters2). The code for the Voter agent is the same as that in part 1 of this tutorial.

Notes on Ascape rules

Voter.java (source)

```
package dave.ascape.voters2;
import edu.brook.ascape.model.*;
import edu.brook.ascape.view.*;
import edu.brook.ascape.rule.*;
import edu.brook.ascape.util.*;
import java.awt.Color;
/**
* An agent for the VoterScape world. Votes according to the opinion of
* a randomly chosen 'moore' neighbour.
*
```

```
	* See also dave.ascape.voters2.VoterScape.java
	*/
	public class Voter extends CellOccupant {
		private int supportedParty = 0;
		private int temp = 0;
		private int numParties = 0;
		public Voter() {}
		public void initialize() {
			super.initialize();
			numParties = ((VoterScape)scape.getModel()).getNumberOfParties();
			supportedParty = randomInRange(0, numParties-1);
		}
		public void scapeCreated() {
			getScape().addInitialRule(MOVE_RANDOM_LOCATION_RULE);
			getScape().addRule(ITERATE_AND_UPDATE_RULE);
		}
		public void iterate() {
			CellOccupant [] neighbors = getHostCell().getNeighboringOccupants();
			int chosenNeighbor = randomInRange(0, neighbors.length-1);
			CellOccupant chosenAgent = neighbors[chosenNeighbor];
			temp = ((Voter) chosenAgent).getSupportedParty();
		}
		public void update() {
			supportedParty = temp;
		}
		public Color getColor() {
			switch(supportedParty) {
				case 0: return Color.blue;
				case 1: return Color.red;
				case 2: return Color.black;
				case 3: return Color.orange;
				case 4: return Color.white;
				default: return Color.green;
			}
		}
		public void setSupportedParty(int p) {
			supportedParty = p;
		}
		public int getSupportedParty() {
			return supportedParty;
		}
		public int getNumParties() {
			return numParties;
		}
		public void setNumParties(int n) {
			numParties = n;
		}
	}
```

Notes on Voter.java

1. scape.getModel().getNumberOfParties().

This line has some interesting features. The number of parties is held by the numberOfParties variable in the root Scape (VoterScape). So the idea is to tell the agent how many parties there are. This way of doing this (rather than, for example, passing the information as a constructor argument) allows the user to change the number of parties at run time.

For the Voter agent to get at the getNumberOfParties() method of the VoterScape class requires the two steps shown: 'scape' and 'getModel()'. What is going on?

scape is protected field of th class, AscapeObject. Since the AscapeObject class is the root of the Ascape system, this fields is inherited by all Ascape objects. It represents the scape (agent) to which the agent in whose code the 'scape' variable appears, belongs.

In the VoterScape example, consider these two lines in the CreateScape() method,

```
voters = new ScapeVector();
voters.setPrototypeAgent(voter);
```

The ScapeVector, voters, "ownes" the voter agent. So, 'scaper' referes to this Scape.

Another line of the code is,

```
addAgent(voters);
```

The votes Scape is added to the root Scape, VoterScape by this instruction.

This is where getModel() comes in. getModel() is a method of the Agent class. It gets the root scape, in this case the Scape owning the voters Scape. This is just the VoterScape class which contains the variables and methods which describe the global properties of the model being designed,