A FRAMEWORK FOR A DYNAMIC INVOCATION INTERFACE AND HIGH-LEVEL INTEROPERABILITY FOR MOBILE AGENT PLATFORMS

by

VAMSI K PUTREVU

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2005

ACKNOWLEDGEMENTS

I would like to first thank my supervising professor, Mr. David Levine, for his constant support and able guidance. I thank him for being patient and helping me in solving various problems, working under him was really “fun”. He has indeed been a great positive influence on me.

I am also thankful to Wahid Bilal and Jimmy, for their useful discussions during the initial phases of my work. I have benefited a lot from Bilal’s research work in analyzing the security model of the Ajanta mobile agent system. I have a special thank you for Mr. Ramesh Danduri who helped me during the documentation phase of my thesis.

I would also like to thank my dad, Mr. P.V.R.K. Ananda Rao, my brother Mr. Raghav Putrevu, for their sincere support and constant encouragement, and above all, I would like to thank my mom Mrs P. Subbulakshmi, her unconditional love has always been my biggest strength.

August 27, 2005

ABSTRACT

A FRAMEWORK FOR A DYNAMIC INVOCATION INTERFACE AND HIGH-LEVEL INTEROPERABILITY FOR MOBILE AGENT PLATFORMS

Publication No. ______

Vamsi Putrevu, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: David Levine

The mobile agent paradigm for distributed systems is inherently suitable for many applications ranging from network management to e-commerce. One inherent problem to its wide spread use is communication between disparate agent platforms and reducing overheads while doing the same. Not all platforms for mobile agents are the same. The Mobile Agent Facility (MAF) proposal is an attempt to standardize this execution environment, this thesis proposes a framework extending the current MAF specification, by which inter-agent communication can be achieved in a more scalable fashion by interpreter based mobile agent systems.

The future of the Internet depends on its ability to transmit video/audio signals over it, the mobile agent paradigm, by its inherent nature saves a lot of bandwidth, and can achieve the above requirement, to corroborate this, a decentralized videoconference using mobile agents

(IBM-Aglets), that aims to dynamically adapt itself to changing network conditions, was implemented.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS……………………………………………………………………………ii

ABSTRACT…………………………………………………………………………………………...iii

LIST OF FIGURES ………………………………………………………………….............xvi

LIST OF TABLES………………………………………………………………… …...xvii

CHAPTER

1. INTRODUCTION.. 1

1.1 BACKGROUND 1

1.2 BASIC CONCEPTS RELATED TO MOBILE AGENTS... 4

1.2.1 AN AGENT EXECUTION ENVIRONMENT 4

1.2.2 AGENT TRANSFER PROTOCOL (ATP)... 4

1.2.3 MESSAGE 4

1.2.4 ITINERARY 4

1.2.5 AGENT LIFECYCLE 5

1.2.5.1 Creation 5

1.2.6 MOBILITY 5

1.2.7 STORAGE.. 5

1.2.8 DISPOSAL 6

1.2.9 INTER-AGENT COMMUNICATION 6

1.3 PROBLEM STATEMENT 6

1.4 ORGANIZATION OF THESIS 7

2. MOBILE AGENTS 19

2.1 INTRODUCTION 9

2.2 AGENTS 9

2.2.1 DEFINITIONS OF AGENT 10

2.2.2 CLASSIFICATION OF AGENTS 11

2.2.3 APPLICATIONS BASED ON AGENTS 12

2.3 MOBILE AGENTS 13

2.4 MOTIVATION FOR USING MOBILE AGENTS 15

2.5 APPLICATIONS OF MOBILE AGENTS 17

2.6 MOBILE AGENT SYSTEM 20

2.6.1 CHARACTERISTICS OF HOSTS (SERVERS) 20

2.6.2 CHARACTERISTICS OF AGENT SERVERS 20

2.6.3 CHARACTERISTICS OF AN AGENT 20

3. AJANTA A REFERENCE MODEL OF A MOBILE AGENT SYSTEM 22

3.1 INTRODUCTION 22

3.2 AJANTA 23

3.2.2 THE AJANTA SYSTEM 24

3.3 AGENT SERVER ARCHITECTURE 26

3.4 AGENT STRUCTURE 29

3.5 NAMING RESOLUTION 30

3.5.1 THE NAME REGISTRY 32

3.5.2 PUBLIC KEY DISTRIBUTION 33

3.6 SECURE AGENT EXECUTION 33

3.6.1 PROTECTION DOMAIN FOR AGENT EXECUTION 34

3.6.2 SECURE ACCESS PROTOCOL 35

3.7 AGENT PROGRAMMING PRIMITIVES 36

3.7.1 PRIMITIVES RELATED TO AGENT CREATION 36

3.7.2 PRIMITIVES RELATED TO AGENT MIGRATION 37

3.7.3 AGENT TRANSFER PROTOCOL 38

3.7.4 PRIMITIVES RELATED TO AGENT COMMUNICATION 44

3.7.5 PRIMITIVES RELATED TO AGENT MONITORING 44

3.7.6 PRIMITIVES RELATED TO FAULT TOLERANCE 44

3.7.7 PRIMITIVES RELATED TO SECURITY 46

4. MOBILITY AND COMMUNICATION 57

4.1 INTRODUCTION 47

4.2 APRIL 3.0 48

4.2.1 AGENT MOBILITY 48

4.2.2 AGENT COMMUNICATION 49

4.3 AGLETS SOFTWARE DEVELOPER KIT 1.0 50

4.3.1 AGENT MOBILITY 51

4.3.2 AGENT COMMUNICATION 51

4.4 D'AGENTS 2.0 53

4.4.1 AGENT MOBILITY 54

4.4.2 Agent Communication 54

4.5 GRASSHOPPER 1.1 55

4.5.1 Agent Mobility 55

4.5.2 Agent Communication 56

4.6 ODYSSEY 1.0 BETA 2 56

4.6.1 Agent Mobility 56

4.6.2 Agent Communication 57

4.7 VOYAGER 2.0 BETA 1 57

4.7.1 Agent Mobility 58

4.7.2 Agent Communication 58

4.8 PLATFORM COMPARISON 59

4.8.1 Agent Mobility 60

4.8.2 Agent Communication 61

5. CORBA AND MOBILE AGENT TECHNOLOGY 74

5.1 INTRODUCTION 64

5.2 CORBA COMMON OBJECT SERVICES 65

5.2.1 Externalization Service 66

5.2.2 Life Cycle Service 66

5.2.3 Naming Service 67

5.2.4 Trading Object Service 67

5.2.5 Security Service 67

5.2.6 Objects by Value 68

5.3 KEY CAPABILITIES OF MOBILE AGENT PLATFORMS 68
5.4 MOBILE CORBA AGENT 69
5.4.1 STRUCTURE OF A MOBILE CORBA AGENT 70
5.4.1.1 Core Object Type 70
5.4.1.2 Bridge Object Type 71
5.4.2 IDENTIFICATION OF A MOBILE CORBA AGENT 71
5.5 MAPPING OF CORBA SERVICES TO MAP CAPABILITIES 72
5.5.1 INITIAL THOUGHTS ABOUT MOBILITY 73
5.5.2 MA SPECIFIC REQUIREMENTS ASSOCIATED WITH CORBA SERVICES 75
5.5.2.1 Externalization Service 75
5.5.2.3 Trading Object Service 75
5.5.3 PRECONDITIONS FOR A MOBILE CORBA AGENT 76
5.5.3.1 Life Cycle Support 77
5.5.3.2 Externalization Support 77
5.5.4 MOBILE CORBA AGENT RELATED INTERACTION SCENARIOS 77
5.5.4.1 Agent Creation 77
5.5.4.2 Agent Termination 78
5.5.4.3 Agent Migration 79
5.6 LIMITATIONS OF THE CURRENT CORBA MODEL 82
5.7 CONCLUSIONS 83
6. MOBILE AGENTS AND AGENT COMMUNICATION 84
6.1 INTRODUCTION 84
6.2 AGENTS AND INTEROPERABILITY 85

6.2.1 THE MASIF STANDARD 86
6.2.3 3—LAYERED PROBLEM OF INTEROPERABILITY 87
6.4.1 ACL AND A GENERIC MESSAGING PROTOCOL 90
6.4.2 ADVANTAGES OF AN ACL 90
6.5 CURRENT ACLS 91
6.5.1 BASIC CONCEPTS OF KQML 91
6.5.2 FIPA AND ITS ACL 93
6.6 BRINGING THE TWO WORLDS TOGETHER 94
6.6.2 INTEGRATING ACL'S AND MOBILE AGENTS 96
6.6.3 PROBLEMS INTEGRATING ACL'S AND MOBILE AGENTS 97
6.7. CONCLUSIONS 98
7. AN INTRODUCTION TO THE MASIF STANDARD 100
7.1 INTRODUCTION 100
7.2 COMMON CONCEPTUAL MODEL 100
7.2.1 INTEROPERABILITY 101
7.2.1.1 Issues addressed by MASIF currently 102
7.2.1.2 Issues to be addressed by MASIF later 104
7.2.1.3 MAF Interoperability Summary 104
7.2.2 BASIC CONCEPTS 105
7.2.2.1 Agent Authority 105
7.2.2.2 Agent Names 106
7.2.2.3 Agent Location 106
7.2.2.4 Agent System 106

7.2.2.5 Agent System Type 106
7.2.2.6 Agent System to Agent System Interconnection 107
7.2.2.7 Regions 109
7.2.2.8 Region-to-Region Interconnection 110
7.2.2.9 Serialization/Deserialization 110
7.2.2.10 Codebase 110
7.2.2.11 Communications Infrastructure 111
7.2.2.12 Locality 111
7.2.3 AGENT INTERACTION 111
7.2.3.1 Remote Agent Creation 111
7.2.3.2 Agent Transfer 112
7.2.3.3 Agent Method Invocation 112
7.3 FUNCTIONS OF AN AGENT SYSTEM 112
7.3.1 TRANSFERRING AN AGENT 113
7.3.1.1 Initiating an Agent Transfer 113
7.3.1.2 Receiving an Agent Transfer 114
7.3.1.3 Class Transfer 114
7.3.2 Creating an Agent 117
7.3.3 Providing Globally Unique Names and Locations 117
7.3.4 Supporting the Concept of a Region 118
7.3.5 Finding a Mobile Agent 118
7.3.6 Ensuring a Secure Environment for 118
7.3.7 COUNTERING THREATS 119
7.3.7.1 Security Service Requirements 120
7.3.7.2 Authentication of Client 120
7.3.7.3 Mutual Authentication of Agent Systems 121

7.3.7.3 Agent System Access to Authenticate Results and Credentials 121
7.3.7.5 Agent Authentication and Delegation 121
7.3.7.6 Agent and Agent System Security Policies 122
7.3.7.7 Integrity, Confidentialit,Replay Detection, and Authentication 122
7.4 CONCLUSIONS 122
8. INTER-AGENT COMMUNICATION 124
8.1 INTRODUCTION 124
8.2 A DYNAMIC INVOCATION INTERFACE 125
8.2.1 DIFFERENT PROTOCOLS FOR COMMUNICATION 126
8.2.2 AN ILLUSTRATION TO EXPLAIN THE PROPOSAL 127
8.3 THE SCENARIO 128
8.3.1 PROBLEMS BEING FACED 128
8.3.2 THE FUNCTIONING OF THE STORE TODAY 129
8.3.3 SOLUTION OFFERED BY STATE-OF-ART TECHNOLOGIES 129
8.3.3.1 Problems with the current solution 131
8.3.4 HIGH LEVEL RESULTS 131
8.3.5 BEHIND THE SCENES 132
8.3.5.1 The MAF services this scenario uses 133
8.3.5.2 Description of low level details 133
8.3.6 LOCATE 134
8.3.6.1 Details of the above action 134
8.3.7 CREATE 135
8.3.8 REGISTER 135
8.3.9 MOVE 136

8.3.10 STATUS 137

8.4 CONTACTING STATIC/UN-REACHABLE AGENT SYSTEMS 138

9. A DECENTRALISED VIDEO CONFERENCE APPLICATION 152

9.1 INTRODUCTION 140

9.2 SYSTEM ARCHITECTURE 142

9.3 CONCLUSIONS 144

10. CONCLUSIONS 146

10.1 FUTURE WORK 147

REFERENCES 148

BIOGRAPHICAL INFORMATION 156

LIST OF FIGURES

2.1 REMOTE PROCEDURE CALLS (RPC)…………………………………………………… 33

2.2 REMOTE EVALUATION (REV) …………………………………………………… 33

2.3 MOBILE AGENT…………………………………………………………………... 34

3.1 THE AJANTA SYSTEM ARCHITECTURE……………………………………………. 44

5.1 A MOBILE CORBA OBJECT…………………………………………………… 69

7.1 AGENT SYSTEM ARCHITECTURE……………………………………………….. 106

7.2 AGENT SYSTEM TO SYSTEM INTERCONNECTION……………………………... 107

8.1 ILLUSTRATION OF PROPOSAL……………………………………………………... 127

8.2 KEY ACTORS IN THE SCENARIO…………………………………………………... 132

9.1 DECENTRALIZED VIDEO CONFERENCE ARCHITECTURE…………………………….. 141

LIST OF TABLES

3.1 AGENT MOBILITY SUPPORT……………………………………………………………… 38

3.2 COMMUNICATIONS AND CONTROL PRIMITIVES……………………………………. 45

4.1 A COMPARATIVE STUDY OF DIFFERENT MA PLATFORMS…………………………… 61

CHAPTER 1
INTRODUCTION

1.1 Background

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 code, and in particular mobile agents, will be an essential tool for allowing such access.

In the computer context an agent is a program that acts for a user, technically an agent is associated with a certain amount of autonomy. With the exponential growth in the Internet user base, network-centric programming and applications have gained a lot of popularity; various driving forces for the above have been the wide use of interpreter-based languages like Java, which facilitates mobility of code across the network. There has been vast use of Internet based technologies in private networks- giving rise to intranets and extranets, thus fuelling the demand for network-based applications.

Along with this growing need for network based applications there seems to be a need for new network computing paradigms that can overcome the barriers posed by congestion and unreliability. The "agent" paradigm offers a promising solution to the above problems. Perhaps the most promising among the various new paradigms is that of *mobile agents.*

This paradigm naturally supports many applications like network management, adaptive network load balancing, on-line shopping, real time control systems, distributed computing and e-commerce applications. With the popularity of the Internet, more efficient ways of representation and filtering of information are necessary.

Applications could specify the list of tasks and different sites to the agent and launch it on the Internet. While performing the tasks, the mobile agent could follow the itinerary already specified or it could determine its path dynamically based on its intelligence or some other requirement. This agent could also spawn other mobile agents for some types of distributed processing. The agents report the results to the owner or some other agent or application.

Mobile agent technology has become popular primarily because of the efficient way it provides for the access and the manipulation of remote information. It is based on a different approach for the design of distributed systems [10, 12]. The traditional client-server and the message-based architectures treat local and remote resources in the same way, in that they use location transparency. From the user's point of view all information is stored locally. In contrast to the traditional technologies, the mobile agent architecture considers that local interaction is more efficient than the remote one. The solution proposed by mobile agents is for software to migrate to remote hosts, where information is stored, so that the user's request can be executed [3]. It means that agents can take decisions or fulfill previously designed operations in an autonomous way on the remote site, even if the home host becomes temporarily unavailable.

The mobile agent paradigm has evolved from the traditional client-server paradigm in which clients communicate through messages. The traditional systems used include

messaging, simple datagram sockets, remote procedure calls (RPC), conversations and remote evaluation (REV) for communication. Several use asynchronous protocols, for example, messaging, and others used synchronous protocols, for example, RPC. Mobile agents use asynchronous protocol. In RPC, only data is transferred in the form of parameters. In REV, the code is sent from client to server and data is returned back. In mobile agents, the data, code and execution context is transferred between clients and servers in a fully duplex manner.

Because the technology and the industry are new a lot of mobile agent systems (for example, Crystaliz's MuBot, Dartmouth college's AgentTcl, IBM's Aglets, the open group's MOA, GMD FOKUS's JMAF/Magna, and General Magic's Odyssey and University of Minnesota's Ajanta), differ widely in their architecture and implementation.

The difference in the mobile agent systems prevent interoperability and rapid proliferation of the agent technology, and has probably impeded the growth of the industry. This paper describes how interoperability is being achieved among various mobile agent systems and discusses the architectural changes that have to be made in a generic mobile agent system to be compatible with the current *common conceptual model* [7].

Mobile agents also raise serious concerns in terms of security. The term mobile means that the agent would have to move to another server and execute there. For its execution, it would need server resources, which exposes the server to dangers of misuse or tampering of resources by agent. Similarly a malicious host could also tamper with the agent.

1.2 Basic concepts related to mobile agents

1.2.1 An Agent Execution Environment

AEE is the agent's workplace. It is a stationary object that provides means for maintaining and managing running agents in a uniform environment regardless of the underlying operating system. The AEE host and other agents are secured against malicious agents, and agents have the ability to authenticate an AEE before they visit it to provide a degree of protection against malicious AEEs. An AEE supports all aspects of the agent lifecycle (described in the next section) and provides certain informational services to the agents. For instance, an agent can query an AEE for its current load.

1.2.2 Agent Transfer Protocol (ATP)

This is defined to enable agent mobility between AEE hosts. When agents travel, their code and state are encapsulated in ATP messages. An ATP *daemon* is a long-running resident program that accepts ATP connections from other AEE hosts on a specified port. An ATP daemon can support multiple *named* AEEs on the same host and port.

1.2.3 Message

This is an object exchanged between agents. It allows for synchronous as well as asynchronous communication. Message-passing is used by agents to collaborate and exchange information in a loosely coupled fashion.

1.2.4 Itinerary

This is an agent's travel plan. It provides an abstraction for travel patterns and routing. An agent can be given an itinerary at the time of its creation, be provided with an itinerary in a message, or modify its current itinerary based on its state, and the state of its current AEE.

1.2.5 Agent lifecycle

1.2.5.1 Creation

Agent *creation* takes place in an AEE. There are two ways to create an agent: *instantiation* of a new agent using agent code in a specified location, or *cloning* of an agent that is currently executing in an AEE. When a new agent is instantiated, its code is fetched from the specified location; it is assigned an identifier, inserted into an AEE, and initialized. The agent starts executing immediately after it has been initialized. An agent can be instantiated by another agent or by a non-agent ATP client external to the AEE in which a new agent is to be instantiated. When an existing agent is cloned, an almost identical copy is produced in the current AEE and the execution restarts in the clone. The only difference is a new agent identifier for the clone. An agent can clone itself or be cloned by another agent.

1.2.6 Mobility

Dispatching an agent from one AEE to another will remove it from its current AEE and insert it into its destination AEE, where it will restart execution. In other words, the agent is *pushed* into its new AEE. The *retraction* of an agent will *pull* (remove) it from its current AEE and insert it into an AEE from which the retraction is requested.

1.2.7 Storage

Deactivation of an agent is the ability to temporarily stop and remove its thread(s) from the current AEE and store the agent code and its current state in secondary storage (typically, in the file system). *Activation* of an agent restores its thread(s) and its state in the AEE. An agent can deactivate it, or another agent, and specify that it should be activated after a certain period of inactivity, or upon receipt of a message of a certain kind(s).

1.2.8 Disposal

The disposal of an agent stops and removes its thread(s) from the AEE.

1.2.9 Inter-agent communication

Agent-specific communication is achieved through *messaging*, which involves sending, receiving, and handling messages synchronously as well as asynchronously. Agents can also communicate with other agents and with objects in the distributed computing environment, subject to security restrictions.

1.3 Problem Statement

Consider a network of a large company. Usually, such a network is connected to the Internet via a router and protected by a firewall running on the router. Thus, only a fixed number of ports are visible to users outside of the company's intranet. A generic mobile agent communicates with the router in the same way as it would communicate with peer agents within the intranet. Security is less of an issue within the intranet. However, the security and communication protocols used by this agent are the same for communicating with router or communicating within the intranet. Thus this mobile agent system is not able to use its locality, and hence the intranet strategy.

The differences among the various mobile agent systems prevent interoperability and rapid proliferation of the agent technology, and have probably impeded the growth of the agent technology; therefore to promote both interoperability and system diversity some aspects of the mobile agent technology must be interoperable.

The present architecture of a generic mobile agent system does not allow communication with other agent systems, thus rendering itself a “dummy resource” within a

host where other agents are residing. Not only is this agent a "dummy resource" to other agents present at the host but also it cannot access other agent services by itself.

This thesis proposes and analyses a framework for a multi protocol based communication service so that a generic mobile agent can make use of its locality and chooses the most efficient protocol.

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 timesaving from avoiding intermediate network traffic is currently less than the time penalties from slower execution and the migration overhead, thus we look into ways by which we can overcome some of these overheads.

1.4 Organization of Thesis

In chapter 1, the author introduced the mobile agent systems, issues involved and the problem statement. Chapter 2 gives the motivation and historical evolution of the mobile agent systems. In chapter 3, the author discusses the Ajanta mobile agent system architecture, which is used as a reference model in the rest of the thesis. Chapter 4 establishes the state-of-art in mobility and communication frameworks of the current mobile agent systems. In chapter 5 the author addresses the interoperability problem from the CORBA standard perspective. Chapter 6 discusses the interoperability problem from a high—level; an analysis of the current agent communication languages along with how they can be used for the current mobile agent platforms is done. Chapter 7 provides an overview

of the MASIF specification for interoperability standards. In chapter 8 the author proposes and analyzes a dynamic invocation interface for mobile agent communication. In chapter 9 the author discusses the design of a decentralized and fault tolerant videoconference, which was implemented using IBM Aglets

Chapter 10 concludes the paper with the contribution of this thesis and identifies areas of future work.

CHAPTER 2
MOBILE AGENTS

2.1 Introduction

This chapter describes the evolution of mobile agents. In section 2.2, various definitions of agents are given along with classification and application of agents. In section 2.3, mobile agents are described in detail. Section 2.4 discusses the motivation for using mobile agents in an open network like Internet. In section 2.5, various applications of mobile agents are given. Section 2.6 gives various characteristics of mobile agent system components.

2.2 Agents

The topic of agents is very broad and interesting and different researchers have defined the agents in their own way depending on their area of research and their application. Here are some of the generic definitions of agents:

- "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon the environment through effectors." [17].
- "Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed."
- "Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy and in doing so, employ some knowledge or representation of the user's goals or desires."[18].

2.2.1 Definitions of Agent

Different vendors and researchers have their own notion and definition of agents. It is interesting to see some of the definitions given later in this page:

The Maes Agent: "Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed." [9,10].

The KidSim Agent: "Let us define an agent as a persistent software entity dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special purpose' distinguishes them from entire multifunction applications; agents are typically much smaller." [11].

The MuBot Agent: "The term agent is used to represent two orthogonal concepts. The first is the agent's ability for autonomous execution. The second is the agent's ability to perform domain oriented reasoning." [8].

The AIMA Agent: "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."

The Hayes-Roth Agent: Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions [12].

The IBM Agent: "Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires." [13].

The Wooldridgep Jennings Agent: "... a hardware or (more usually) software-based computer system that enjoys the following properties [14]:

Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;

Social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language.

Reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;

Pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative."

The SodaBot Agent: "Software agents are programs that engage in dialogs [and] negotiate and coordinate transfer of information."

The Brustoloni Agent: "Autonomous agents are systems capable of autonomous, purposeful action in the real world." [15].

2.2.2 Classification of Agents

As can be seen by the number of definitions of the agents, it would be nice to have a generic classification and definition of the agent based on its function. Some generic definitions are given later in this page [16]:

- Reactive Agents (Sensing and Acting)—these respond in a timely fashion to changes in the environment.
- Autonomous Agents—these exercise control over their own actions.
- Goal oriented (Pro-Active or Purposeful)—these do not simply act in response to the environment.

- Temporally Continuous—these are continuously running processes.
- Communicative (Socially Able)—these can communicate with other agents and sometimes with people.
- Learning (Adaptive)—these agents are capable of changing their behavior based on it's previous experiences.
- Mobile Agent—can transport itself from one machine to another.
- Flexible—these agents do not have scripted actions.
- Character—these have believable personality and emotional state.

Almost all the agents have the first four characteristics and the other characteristics are based on their particular functionality. For example mobile agents are reactive, autonomous, goal oriented and temporally continuous but the main characteristic is its mobility, which distinguishes it from a learning agent, although a mobile agent could also be learning as well.

2.2.3 Applications Based on Agents

The agents are used for a number of applications, some of which are listed later in this page [19,20]:

- Artificial Intelligence—Agents are used extensively in the field of artificial intelligence. It improves collaborative online social online social events. Agents are also used in the field of robotics.
- Distributed Systems—Agents have some advantages over the traditional client/server architecture, which are used to enhance the client/server architecture. Agents reduce client/server network bandwidth problems by moving a query from the client to server. They allow decisions about the location of code to be made at the end of development cycle, when a lot more is known about the performance of the

application, reducing the design risks. The agents can work even when the network goes down, they can work offline and return the results when the network goes back online.

- Network management—The agent reduces network bandwidth by transferring the queries by transferring the query or transaction from the client to the server, thus minimizing bandwidth. An agent can be programmed to visit several nodes in the network in succession carrying out some task, this ensures that communication is defined clearly and the designer need only concentrate on the tasks to be assigned to an agent. Another advantage is the agent can work offline or its transaction can be stalled at one site and it can resume at the other site where it was stalled. The agents also provide real time notification where an agent situated at any remote site may notify the host of an important event immediately. It also allows parallel execution where large computations can be divided depending on resources.

With a background in agents, the author describes mobile agents in detail.

2.3 Mobile Agents

An agent is defined as a self-contained software element that acts autonomously on behalf of a user (e.g., person or organization) [5, 7]. Each agent has its own thread of execution, which means that it can perform tasks on its own initiative. A mobile agent has, in addition, the unique ability to migrate from one host in a network to another [5, 7]. The definition of a mobile agent contains at least two key words that raise important issues. The first key word refers to the autonomy of a mobile agent. This is a feature that allows the agent to act on its own by using the data and the mobile logic which it incorporates, and does not need human intervention or guidance. The user can perform other activities in the

meantime. Further, the time needed to fulfill the tasks is reduced because interaction with the user is avoided. The agent is designed to deal with any situation that may occur during execution. There are however cases when the lack of human supervision may lead to unexpected and undesired actions, such as the violation of the private information associated to the agent's owner [5]. The relation agent-owner should be based on trust because the agent has access to information stored in the user's profile, information that may be private.

Due to the fact that an agent can collect, manipulate, and distribute data, including user's personal data, it is possible that there is a violation of privacy when the agent communicates and/or exchanges data with its environment and/or agents belonging to different systems. In order to defend the user's privacy, Privacy-Enhancing Technologies (PETs) (e.g., Identity Protector) [6] should be added or integrated in mobile agent systems.

The second key word in the definition refers to the mobility of an agent. The mobility feature enables the agent to travel to the host where the data is physically stored. In this way the transfer of a large amount of data in the network is prevented. The agent accesses the data locally and so provides only the results to the user. There are at least two advantages to this approach: low traffic in the network and a better use of network resources. The migration of an agent within the network should respect rules concerning the agent transport, agent interaction and security, otherwise it is not possible to ensure interoperability among mobile agents which belong to different vendors' systems.

The security issue has always been considered to be the weak point of mobile agent technology. The security problem regards the security of the host that may receive a potentially malicious agent, as well as the security of the agent that should be protected from potentially malicious hosts. Closely related to this aspect, is legal responsibility, which

concerns authentication of mobile agents, secrecy, privacy, data quality, and many other similar aspects. Questions such as: “Is the agent really what it pretends to be?” or “Has it the right to access this kind of data?” should have a clear response because otherwise the host will not know how to react to the agents’ requests.

2.4 Motivation for Using Mobile Agents

Traditional distributed applications have relied on the client/server paradigm, where the main means of communication is either through message passing or by Remote Procedure Calls (RPC), This way of communication is usually synchronous (i.e., the client sends the request and waits for the server and waits for the response from the server with the result). Stamos and Gifford proposed an alternate architecture called Remote Evaluation (REV) in 1990 [22]. In REV the client instead of invoking a remote procedure, sends its own procedure code to the server and requests the server to execute it and return the results. Earlier systems like the R2D2 [25] and CHORUS [26] introduced the concept of active messages that could migrate between the different nodes in a network, carrying the code to be executed at the nodes. A more generic concept is a mobile object, which encapsulates data along with a set of operations on that data, and which gets transported between the nodes in a network. EMERALD [27] was an example of an early system that provided object mobility but it was limited to homogeneous local area networks. The Mobile agent paradigm has evolved from these antecedents. Figures 2.1, 2.2 and 2.3 illustrate how mobile agent paradigm differs from its antecedents. In RPC data is transmitted between the client and server in both directions. In REV, code is sent from the client to the server and data is returned. In contrast a mobile agent is an encapsulated program that is sent by the client to the server, But unlike a procedure call the, it does not have to return the results to the client,

but instead it could migrate to other servers, transmit data back to its origin or migrate back to it's origin if appropriate.

Telescript [28], developed by General Magic is one of the first systems expressly designed to support mobile agents in commercial applications. It was followed by other systems like Tacoma [29] and Agent Tcl (D`Agents) [] where the agents were written using script languages. The emergence of Java with its support for mobile code has lead to development of other systems such as Aglets [30], Voyager [31], Concordia [13] and Ajanta [14].

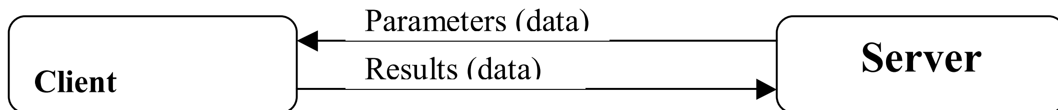

Figure 2.1 Remote Procedure Calls (RPC).

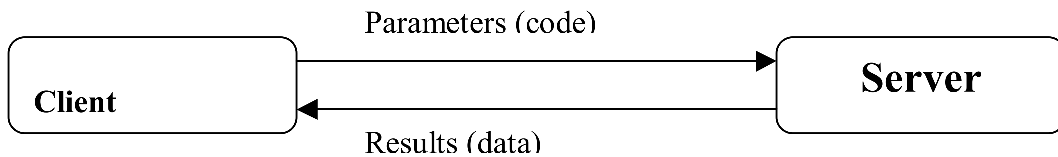

Figure 2.2 Remote Evaluation (REV).

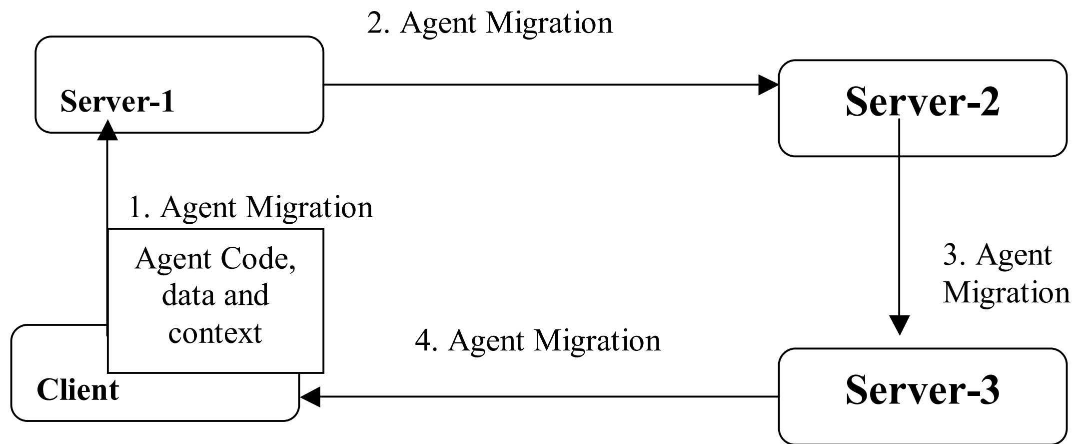

Figure 2.3 Mobile Agent.

2.5 Applications of Mobile Agents

The exponential growth of the World Wide Web and the advances made in the field of Mobile Agents has led to the development of many applications based on Mobile Agent paradigm. Some of the agent applications are discussed later in this page.

Information search and filtering applications often download and process large amounts of server resident information, and generate comparatively small amounts of result data. Instead, using mobile agents, which execute on server machines and access server data without using the network, the bandwidth requirements can be reduced.

Some applications involve repeated client/server interactions, which require either maintaining a network connection over an extended period, or making several separate requests. If mobile agents are used instead, the client does not have to maintain a network connection while its agent's access and process information, as the agent can act autonomously based on the data it collects dynamically. This permits increased asynchrony between the client and server. This feature is especially useful for mobile computers, which typically have low bandwidth, unreliable connections to the network, and are often switched

off to save on power consumption. Also, the repeated client/server interactions are reduced to two agent transfer operations, thus reducing the frequency of network usage as well.

In client/server applications, servers typically provide a public interface with a fixed set of primitives. Clients may need higher level functionality composed of these primitives, and their requirements can change over time. Rather than modifying the server interface to support such requirements for every client, a client can maintain its own interface at the server node, using a mobile agent. This has the added advantage of reducing the number of network-based interactions required. Service providers to dynamically enhance server capabilities can exploit the same feature.

Mobile agents can also be viewed as a mechanism for introducing parallel activities, since they execute concurrently. A client can decompose its task among multiple agents for providing parallelism or fault tolerance. It should be noted that when the client is decomposing the tasks there are no resource conflicts.

Mobile agents support real time system control [41]. Consider an application that uses RPC's to control a device; it may be difficult (if not impossible) to guarantee that it will meet the real time deadlines associated with the device. This is because communication delays are not accurately predictable, unless the underlying network provides quality of service guarantees. Instead, the application can send an agent to the device and control the device locally, resulting in better predictability.

An active mail message is a middleware program that is based on mobile agents. An active message service is a program that interacts with its recipient using a multimedia interface, and adapts the interaction session based on the recipient's responses. The mobile agent paradigm is well suited to this type of application, since it can carry a sender defined session protocol along with the multimedia message.

Agents have been used in the field of E-Commerce. Vendors can set up online shops, with products, services or information for sale. A customer's agent would carry a shopping list along with a set of preferences, visit various sellers, find the best deal based on the preferences, and purchase the product using digital forms of cash. This application imposes a broad spectrum of requirements on mobile agent systems. Apart from mobility, it needs mechanisms for restricted resource access; secure electronic commerce, protection of agent data, robustness and user control over roving agents.

Agents are used in applications that need to monitor events on remote machines also benefit from mobile agents, since agents need not use the network for polling. Instead of periodically downloading the required data, an agent can be sent to the access the data. The agent can inform the user when a specified event occurs.

Manufacturers of cell phones, personal organizers, car radios and other consumer electronic goods are all introducing new features and are becoming the primary focus of Mobile agent developers. Since these devices are not online completely and they can use the property of the mobile agent, an agent once launched can operate autonomously of its host that launched it.

Besides the above mentioned applications agents some of the other applications that have been developed based on the mobile agent paradigm are network maintenance, testing and fault diagnosis, installing and upgrading software on remote machines. A file sharing system and a Calendar manger has been developed based on the Ajanta a java based mobile agent system.

2.6 Mobile Agent System

The mobile agent system consists of hosts and agents. The host also acts as an agent server to provide an agent an execution environment. The characteristics of each entity in the system are listed later in this page:

2.6.1 Characteristics of Hosts (Servers)

- An agent host must allow multiple agents to co-exist and execute simultaneously.
- Provide means of communication between the agent and host and between agents.
- Provide a transport mechanism to transfer agents to other hosts and receive agents from other hosts.
- Since and agent takes with it its state of execution the host must be capable of saving the state of an executing agent and transmit it, conversely it must be capable of receiving an agent and resume execution from the point it was interrupted.
- The host must ensure that the various agents executing on it don't interfere with each other.

2.6.2 Characteristics of Agent Servers

- It provides an environment for uniform execution and maintenance of agents on the server.
- Secure the host from malicious agent attacks.
- Prevent an untrusted agent from accessing sensitive data and resources, while authorizing a trusted agent to access various allocated resources.

2.6.3 Characteristics of an Agent

- An agent has a unique identity on a server.
- An agent must have means of determining messages other agents send and must be capable of sending messages to other agents.

- An agent must be tactile (i.e., Mobility and persistence).
- An agent must be social, (i.e., it must communicate and collaborate with other agents).
- An agent must be cognitive (i.e., it must adapt to a situation, it must learn from it's past and must be goal oriented).
- An agent takes with it its state of execution.
- An agent is autonomous (i.e., once it leaves it's creator it can be trusted to make decisions on it's itinerary and what it will execute).
- An agent can receive input from external sources and make decisions base on the input regarding their itinerary or execution, independently without any external influences.

Having introduced the mobile agent concept, the author analyzes the prior work done in this area in the next chapter i.e., chapter 3.

CHAPTER 3
AJANTA A REFERENCE MODEL OF A MOBILE AGENT SYSTEM

3.1 Introduction

This chapter is an introduction to Ajanta a Mobile Agent system, developed in University of Minnesota, by Dr. Anand Tripathi. This system is a Java Based Object oriented system for programming and developing Mobile Agents. The Mobile agent programming in Ajanta is based on the concept of network mobile objects, Agents in this system are active mobile objects, which not only encapsulate the code, but also include the data to be executed with the code. Ajanta's programming environment is based on a set of primitive operations for agent creation, dispatch, migration and remote control. Agents can access server resources using a proxy based access control mechanism This architecture provides mechanisms to protect server resources from malicious agents, agent data from tampering by malicious servers or communication channels during its travel, and protection of name service data and the global namespace. Ajanta uses a proxy-based mechanism for secure access to server resources by agents. Using Java's class loader model and thread group mechanism, isolated execution domains are created for agents at a server. Ajanta uses generic authentication protocol to conduct client/server communications when protection of resources is required. In this chapter the author gives a description of the system architecture, agent structure, agent server structure, naming policy followed by Ajanta and the programming primitives of Ajanta for activities including agent creation, agent transfer protocol, agent execution on a server and communication between the agents. Section 3.2 gives the requirements Ajanta was based on and a general overview of Ajanta.

3.2 Ajanta

Ajanta is a mobile Agent System developed at the University of Minnesota. In section 3.2.1, the system requirements for Ajanta are elaborated. 3.2.1 Requirements for Ajanta

Security and robustness are among the most important requirements of an agent programming environment. In order to build any agent based application, a set of hosts have to run a facility that receives the visiting agents and provides to them an execution environment and access to its services. These issues are referred to in the Ajanta system architecture.

Servers willing to host agents originating from unknown sources require a high level of confidence in the protection mechanisms to enforce the desired security policies for preventing unauthorized or malicious agents from leaking, destroying or altering the server's resources or disrupt its normal functioning. Security mechanisms are thus necessary to safeguard host’s resources. Conversely, an agent may need to be protected from its host server, because it may carry sensitive information about the user it represents. This includes the requirement of preserving the integrity of the information collected by an agent during its visit to various hosts.

Robustness of the system is also a concern, especially in open, unreliable environments such as the Internet. A user can have a high level of confidence in his agent based applications only if he can monitor and control the agents at all times. Therefore, an ideal agent programming system should provide suitable mechanisms using which a user could monitor the roaming agents’ status and control them remotely, irrespective of their location on the network.

A mobile agent system must, in addition, present a programming framework that is intuitive and powerful. It must provide the programmer a set of primitives that allow the

user to partition the application's tasks among agents, specify their migration plans, control and communicate with them as they traverse the network, and so forth. This task is significantly simplified if a uniform, location independent global naming scheme is used, so that the programmer can refer to agents, servers and resources using location independent identifiers.

The initial step in creating any agent-based application is to define the services to be provided to an agent by its host servers. Then the server needs to be implemented with appropriate resources to provide those services, an agent server also needs to support a generic set of services to allow an agent to migrate from one server to another, verify an agent's identity, create an execution environment for an agent, and grant an agent controlled access to its local resources. It should be possible to dynamically change the access control policies that a service owner may wish to enforce on visiting agents. It should also be possible to add new services with time. Thus, a mobile agent programming system needs to provide customizable agent servers for hosting visiting agents, and a set of primitives for the creation and management of agents. In section 3.2.2, Ajanta mobile agent system is explained.

3.2.2 The Ajanta System

The Ajanta project was motivated mainly to investigate an agent programming system architecture to address the requirements stated. In Ajanta, the mobile agent implementation is based on the generic concept of a mobile object [42]. Agents are active mobile objects, which encapsulate code and execution context along with data. Ajanta is implemented using the Java [43] language and its security mechanisms [44]. It also makes use of several other facilities provided by Java, such as object serialization, reflection, and remote method invocation.

Ajanta offers a programming environment, which reflects several useful and novel features. For the ease of building new applications, Ajanta provides implementations of a generic agent and a generic agent server. The Agent class of Ajanta defines the generic agent, and the AgentServer class defines the generic agent server. These two classes together implement the basic functionalities and protocols of the Ajanta architecture. These classes can be suitably extended for building agents and servers required by an application. The generic agent server can control a visiting agent's access to the host resources at any desired level of access control granularity. This policy is based on the Agent's credentials and it can be dynamically altered. Ajanta also provides mechanisms by which any tampering of an agent's data can be detected [45]. A novel feature supported by Ajanta is the composable pattern of migration for building agent itineraries. A pattern separates the specification of an agent's migration path from its computation tasks. A pattern can also include suitable exception handling mechanisms to deal with migration related exceptions. The basic migration primitive in Ajanta allows an agent to specify migration to a specific host or collocation with another agent or some globally named resource.

Ajanta architecture uses a location independent global naming scheme for referencing or communicating with agents, servers, and any other resources. Here one can refer to all agents and servers without any knowledge of their locations. The name service is also used for distributing public keys of the users and other entities such as the servers and the agents.

Agents collocated at a server can communicate with each other using method invocations or shared objects. Moreover, an agent can communicate with a remote agent using Java's RMI facility. In this context, if required, an Ajanta server is given mechanisms

to deny an agent access to the RMI facility. Any RMI based client/server interaction in Ajanta can be authenticated if desired by the server.

For high confidence, Ajanta provides facilities for applications to monitor the status of their roving agents, and control them remotely. If needed, an application can abort the execution of its remote agents. Ajanta enforces appropriate security in the execution of these control primitives, which can change an agent's travel plans. For error recovery, Ajanta provides a model for handling exceptions encountered by an agent. It introduces the concept of a guardian object associated with one or more agents of an application to handle those exceptions for which the agent is unable to perform any recovery. In section 3.3, Agent Server Architecture is presented.

3.3 Agent Server Architecture

In Ajanta each agent server conceptually contains two entities – agents and resources that the agents can access. Multiple agents can be running on a host each running in its own protection domain. The server maintains a domain registry to keep track of the agents currently executing on it.

The agent server has an agent environment component; this object acts as an interface between the visiting agents and the server. When an agent arrives at the server it is provided with a reference to this object. They then communicate with the server through the agent environment’s interface. This interface allows the agents to access resources, request migration and so forth.

The server also provides application specific resources. A resource is an object that acts an interface to some service or information available on the host. The server maintains a resource registry, which is used in setting up safe bindings between he resources and agents.

The Agent Transfer component implements the Ajanta's Agent Transfer protocol that allows agents to migrate between the hosts. When an agent requests a transfer this module contacts the corresponding module on the destination server and transfers the agent across the network.

The final component is the server interface, which is remotely invocable method or interface. It supports status queries about the agent resident on it; it also provides primitives for tasks such as terminating agents, recalling them to their home sites and so forth. The system architecture is as shown in figure 3.1

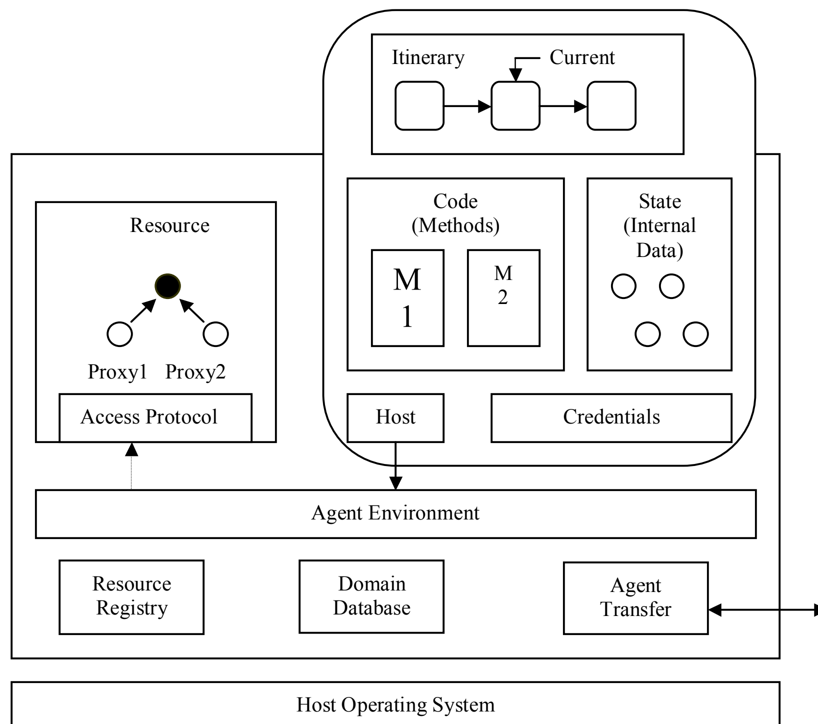

Figure 3.1 The Ajanta System Architecture.

Some of the important elements of Ajanta's computational model are described later in this section.

Principal: Actions in the system are always performed on behalf of some authorized principal, an entity that has a unique identity in the system. Agents, hosts, agent servers, and human users are some of the principals in the system. For each principal, the Ajanta system uses DSA keys for signature and El-Gamal keys for encryption.

Creator: The agent is usually created by another principal, such as an application program, or another agent—Ajanta calls this the creator of the agent.

Owner: This is the human user whom the agent represents. An agent's owner is different from its creator.

Guardian: An application assigns to each of its agents a guardian object, which is responsible for dealing with exception conditions encountered by the agent. If the agent encounters any exceptions during its execution, it is transported to its guardian, which takes the appropriate recovery actions. An agent's creator can also act as its guardian.

Agent Credentials: This is a signed certificate carried by each agent describing its identity and origin. Its tampering can be detected. It contains the names of the agent, its owner, creator, and guardian. Other information encoded into the credentials includes a digest of the "intentions" for which the agent is created. The inclusion of a digest of the intentions places restrictions on the rights granted to an agent by its creator. Typically, the intention is abstracted into an itinerary.

Code Base Server: Other information included in the credential is the URL for agent's code base server, which is a server that provides the code for the classes required by the mobile agent. Typically the creator of an agent would act as its code base server.

3.4 Agent Structure

An agent is a Java object it consists of code—represented by a set of methods in its Java class and state (i.e., the data members of its class). The basic Agent structure is defined in the Agent class. Ajanta implements mobility of agents using Java's object serialization feature. The concept of Mobile Agent implies that a thread carry with it it's state of execution this cannot be implemented using an object serialization, since it cannot capture the execution of the thread. This problem is overcome in Ajanta by making the threads lightweight, It does not capture the state of the thread, no code is transported on an Agent's migration, All code migration in Ajanta takes place "on demand" during an agent's execution. A server associates a system defines class loader with each agent. This first checks the servers class path for the classes if not found it contacts the agents code base server.

An agent carries with it a set of credentials as a part of its state. The credential help identify the agent for access control purposes. The agent has a global name assigned in the URN formats. Apart from this name the credentials also contain identities of the agents owner, creator and guardian. The owner is the human user whom the agent represents. Access control policies are usually specified in terms of owner's authority and privileges. The creator may be different from the owner; it may be another agent run by the same owner or another application. The guardian is an optional part of the credential; usually it is the creator who is the guardian. All errors and exceptions encountered by the agent will be reported back to the guardian. An agent' credentials is usually protected from any tampering by computing the digital signature of its owner on the owner and including it within the object.

Another component of the credentials is a code base for the agent. This is a trusted server process, which has access to all the Java classes that the agent may require, Usually

the application runs the code server as a part of the agent server itself. If the agent's current host requires some class it contacts the agents code base and uses a secure, authenticated protocol to load the class.

An agent contains four kinds of data. The first kind consists of data defined by the base agent class, which can get modified during the agent's migration. The second kind of data is aggregated in a collection called ReadOnlyContainer; any tampering of this data can be detected. This contains some system defined agent specific data and also any application defined constants. The third kind consists of encrypted objects intended for some specific servers, these are put in a container called TargetedState, and Secrecy of this data is an important requirement. The fourth kind of data consists of objects that an agent gets from a host dynamically and carries it along. It is important to protect this data from deletion or tampering by other servers it visits later. The agent checks in an object into this container, the object is then cryptographically protected so that any attempt to modify it or delete it will be detected.

The agent sometimes also contains an itinerary object. It encodes the travel path of an agent along with the code to be executed at each destination. The itinerary is usually assigned to the agent by its creator, but it is modifiable by the agent itself in transit.

3.5 Naming Resolution

In a Mobile Agent System the various system entities need to be assigned a unique name, as they have to be identified uniquely when executing in a host. An agent must be uniquely named so that its owner can identify and control it anywhere in the network. The servers need to be uniquely identified so that the agents can specify their destination. Some namespaces may be common though, Agents and Agent Servers can share a common name, so that the agent can specify whether it wants to migrate to a particular server or if it wants

to share a co-location with another agent to share information. Ajanta uses global, location independent names that do not change when the entity is relocated. This requires the provision for a name service, which maps a symbolic name to the current to the current location of the named identity.

In Ajanta, following the requirements mentioned, it uses a location independent Uniform Resource Naming (URN) scheme. A URN is a location independent identifier which can be used to access an entity or for querying some of its characteristics. Each URN consists of a namespace identifier (NID) followed by a namespace specific string (NSS). In Ajanta the NID is called the ans, Ajanta Name Space, All the entities in Ajanta are assigned URN's within this namespace, therefore all the URN's have the same urn:ans:<NSS>

The name service framework has to address three security requirements: (1) access control on the name server's registry entries, (2) server authentication, and (3) server replication to protect against "denial of service" attacks. To meet these requirements, Ajanta name service protects its database using access control lists for its entries. The name service maintains DSA public key and the El-Gamal public key for each registered principal, and it executes a challenge/response based authentication protocol [46] for any update operation. An example of a URN in Ajanta's namespace is:

urn:ans:cs.umn.edu/UserName/ResourceName

Here 'ans', is the Ajanta Namespace Identifier [54] and the name space string is 'cs.umn.edu/UserName/ResourceName'. Ajanta uses the existing resolution framework of the Domain Name System (DNS) for URN resolution—the translation of URN to a URL, or another URN, or other characteristics. Here, 'cs.umn.edu' is the DNS domain where the object was first created. Ajanta refers to this as the creation domain, which is used as a hint to locate an object. Within this domain, 'UserName' is a naming authority or a subdomain,

and ‘ResourceName’ is a unique string in this subdomain. This hierarchical structure provides ease of maintenance and delegation of namespaces. The naming authority can create a new name only its name space. Ajanta's name service maintains the mapping between the URN of an entity and its characteristics, including its current location. Application level primitives are provided for the client to create new entries or query/update an existing one. All updates require authentication, and they are performed only if permitted by the access control policies. The URN resolution procedure used in Ajanta first queries the local nameserver. If that name server is unable to resolve locally, it queries the nameserver in the URN's creation domain. The creation domain nameserver maintains the entire URN’s created in its domain and contains the authoritative versions of their name registry entries. In the current version, whenever an object migrates, its creation domain is updated. If an object migrates to a domain other than its creation domain, then both the current domain as well as the creation domain nameservers is updated. Each domain has a name registry, which is a server process for maintaining the resolution of data for the entire URN’s created in that domain. The name registries of different domains interact with each other in order to implement replication. Section 3.5.1 describes the Ajanta Name registry.

3.5.1 The Name Registry

The name registry in Ajanta is implemented based on the Java RMI-based server process. It provides for three basic interface functions: bind, rebind and lookup. The bind primitive sets up the initial association between the URN and a specified registry entry, which includes the current location of the entity among other attributes. The rebind primitive changes the binding of a URN to a new registry entry. The Lookup primitive returns the registry entry of a particular entity for a given URN. Depending on the entity the name registry contains different entries that can be registered.

Some of the entries associated with different entities are often distinct. In an Agent server's entry Ajanta stores the URL location at which it can be contacted for agent migration. With an agent it stores the URN of the agent server on which it is currently residing or executing. For resources, since they can be replicated and multiple copies might exist Ajanta stores a list of agent server URN's at which the resource can be found. The only attribute, which must always have an entry for every entity is its ownership attribute, this determines who is permitted to make critical changes to the entries. Section 3.5.2 describes the role of the name registry in Public Key distribution.

3.5.2 Public Key Distribution

Each entity in the system Agents, Agent Servers, Resources and users and so forth. have public keys associated with them. These keys are stored as a part of the name registry entries. The Name registry is responsible to distribute the public keys on request, Hence the name registry also acts as a public-key infrastructure for the system. This binding between the URN and the public key is critical for the authentication process, since the system cannot always depend on network communications for this purpose.

Public key cryptosystems are used to serve two purposes in Ajanta, Digital signatures and encryption. The name service provides a lookup method in its interface, which allows the clients to determine the public key for a given URN. The reply is signed by the name service itself to prevent tampering.

3.6 Secure Agent Execution

The major challenges in agent execution are, the creation of a protection domain in which to isolate the agent, and the provision of a secure resource access protocol. In section 3.6.1, the author gives detail of the creation of securer protection domains for Agent execution.

3.6.1 Protection Domain for Agent Execution

Ajanta uses two Java mechanisms for implementing protection domains: thread groups and class loading. A thread group in Java is a simple collection of threads. When an agent arrives, a new thread group is created for it. A single thread is created in this group, and is assigned the task of executing the method requested by the agent. During its execution, the agent may create other threads, but it is constrained to create them within its own thread group. Thus, any thread executing that agent's code can be identified by its thread group id. The agent server maintains a domain database indexed by thread group ids, in which it stores the agent's URN and credentials, as well as a reference to the agent object itself. This entry is used by the server's code whenever it needs access to the agent or its identity, when an agent attempts to acquire a resource; its credentials are needed to determine the level of access permitted.

Ajanta uses Java's class loader mechanism to protect agents from each other. The Java virtual machine associates with each class, the class loader instance that loaded it. Two classes loaded by different loaders may have the same name, but are considered different types. This implies that objects of the two classes are not type compatible, even if the classes are in fact identical. In effect, a class loader defines its own namespace. Each agent is assigned a unique class loader instance. Whenever the agent's code encounters an object reference whose class is not currently loaded, the Java virtual machine invokes the agent's class loader, and provides it the name of the required class. The class loader first searches the server's classpath a set of directories on the local file system which contain classes trusted by the server. If the required class is found on the classpath, it is loaded into the virtual machine. Otherwise, the agent's code base is contacted and asked to supply the missing class. This communication may be encrypted and digitally signed to ensure the

integrity of the downloaded code. Since each agent has its own class loader, a malicious agent cannot replace any other agent's classes with its own impostor versions.

Agent Activation

When an agent transfer connection is initiated, the ATP handler thread first reads the transfer request message from the sender. This contains the credentials of the agent being transferred. The handler consults the server's security policy to determine whether to allow the transfer to continue. If the agent is not permitted to execute on the server, the connection is aborted. Otherwise, the ATP handler creates a thread group, a new thread in that group, and a new class loader for the agent. An entry for the agent is made into the domain database. The newly created thread uses a bootstrap mechanism to transfer control into the new class loader's namespace. It then deserializes the agent object from the incoming connection, and sets the agent's host reference to the local server's agent environment. Finally, it transfers control to the agent, by invoking the method in the transfer request. Section 3.6.2 discusses the Secure Access protocol.

3.6.2 Secure Access Protocol

Agents need access to both systems level and application defined resources as they execute. The server must grant access only to authorized agents, and then only to the extent permitted by their privileges. System level resources are protected in Java using the security manager, which encodes security policy. All Java library calls, which access system resources, first invoke the security manager to screen the access. For application level resources however, burdening the security manager further is avoided, since every new resource added to the system would necessitate extending the security manager, making it a large, uncohesive module prone to errors. Therefore, a mechanism is needed, which allows the agent server to provide a secure language level binding between agents and resources.

Each resource must be allowed to define its own security policy, and control its implementation.

Ajanta uses a proxy-based mechanism to access server resources. The resource request and the resource access protocols are discussed in detail in chapter 9.

3.7 Agent Programming Primitives

The primitives for programming in Ajanta are used for controlling the agents in the network. The primitives provide methods for creating and dispatching the agents, control the agent mobility, monitor them, check for exceptions and recover from exceptions.

3.7.1 Primitives Related to Agent Creation

An agent creation primitive allows the programmer to create instances of agents, thereby partitioning the application's task among its roving components. This also introduces concurrency into the system. Agent creation involves the submission of the entity to be treated as an agent, to the system. This could be a single procedure to be evaluated remotely (as in REV), a script, or a language level object. In object-oriented systems, an agent is usually created by instantiating a class, which provides the agent abstraction. The system can inspect the submitted code to ensure that it conforms to the relevant protocols and doesn't violate security policy. Based on the identity of the agent creator, a set of credentials for the agent may also be generated at this time. These are transmitted as part of the agent, to allow other entities to identify it unambiguously.

A newly created agent is just passive code, since it has not yet been assigned a thread to execute it. For activation, it must be dispatched to a specific agent server. The server authenticates the incoming agent using its credentials and determines the privileges to be granted to it. It then assigns a thread to execute the agent code. A variant of the creation

primitive allows an agent to create identical copies of itself, which can execute in parallel with it, and potentially visit other hosts performing the same task as their creator. Another variant is forking of agents, in which the newly created agent retains a parent child relationship with its creator. This allows programmers to create agents that inherit their ownership, privileges, and so forth. from their parents. Section 3.7.2 discusses the primitives related to Agent Migration.

3.7.2 Primitives Related to Agent Migration

When an agent is executing, the agent program may determine that it needs to visit another site on the network. To achieve this, it invokes a migration primitive. The agent server must suspend the agent's execution, capture its state and transmit it to the requested destination. The server at the destination can then receive the agent state and activate it after the appropriate security checks are passed. The destination specified by the agent can either be absolute, (i.e., the name of the server it needs to migrate to), or relative, (i.e., the name of another agent or resource it needs to collocate with). Most systems provide absolute migration primitives. Ajanta supports Relative migration. Some systems build upon their migration primitives to provide higher levels of abstraction, such as an itinerary, which contains a list of servers to visit, and the corresponding code to execute at those locations. Table 3.1 summarizes the basic mobility features provided in the various Mobile Agent systems.

Table 3.1 Agent Mobility Support

System	Naming	Agent Migration
Telescript	Location dependent (based on DNS).	Both absolute *(go)* and relative *(meet)* migration.
Agent Tcl	Location dependent name based on DNS and optional symbolic alias.	Only absolute, using *agent_jump* primitive. The *agent_fork* sends a clone agent instead.
Aglets	URL's based on DNS names.	Only absolute, using the dispatch primitive. Supports *Itinerary* abstraction.
Voyager	Location Independent global ID, as well as local proxy.	Single primitive *(moveTo)* supports both absolute and relative migration.
Concordia	Location dependent (based on DNS). Directory Services available	Only absolute, based on the contents of agent's *Itinerary*.
Ajanta	Location Independent global names.	Single primitive *(go)* supports both absolute and relative migration. Supports *Itinerary* abstraction.
Tacoma	Location dependent (based on DNS).	Single primitive *(meet)* supports both absolute and relative migration.

3.7.3 Agent Transfer Protocol

The generic agent server in Ajanta, which can be extended by a user to define an application specific server, supports the Agent transfer protocol, which is used for agent migration between agents and servers.

An agent requesting migration specifies a destination and the method to be executed there. The interactions between two agent servers to transfer an agent consist of two phases. In the first phase, the current server first sends a request message to the destination server containing the agent's credentials, agent owner's signature for the credentials, a method specification, and some other parameters controlling the transfer itself, including flags indicating whether the transfer should be encrypted and signed, size of the agent and so forth. The credentials identify the agent, thus allowing the destination server to decide whether to permit the transfer. The destination server then verifies the credentials against the owner's signature. If the verification is successful, a positive acknowledgment is sent to sender; otherwise, an exception is sent. On receiving a positive acknowledgment, the sender sends the serialized agent object to the destination.

Two Java mechanisms are used for isolating agents from each other at a server thread grouping and class loading. When an agent arrives at a server, a new thread group is created; all threads created by the agent are constrained to be within this group. Thus at runtime, the actions of an agent's code can be identified by the thread group id. This mechanism also enables the server to distinguish between the agents executing on it, so that they do not interfere with the others working. Ajanta uses Java's class loader mechanism to isolate agents. Each executing agent is assigned a separate Ajanta defined class loader, which is responsible for locating and loading any classes needed during the agent's execution. Each class loader defines a separate name-space for the classes that it loads and prevents an agent from bringing in any untrusted code for security sensitive operations. This class loader ensures that all trusted classes are always loaded from the server's class path; only when a class is not found on this class path, will the class loader look for that class at the agent's code base server.

At the destination server, a new thread group and an Ajanta defined class loader are created for this new agent's execution. Also a new server thread is created in this thread group to receive the serialized agent and deserialize it, and to execute the method specified in the transfer request. The Startup class does the deserialization of the agent. This ensures that a class loader loads all classes of the arriving agent, which is separate from the system class loader. Moreover, as a class loader is exclusively created for each agent, classes for all agents are kept under separate namespaces. If the transfer is successful, the thread created to handle the agent transfer first executes the arrive method of the agent, then it executes the method specified in the transfer request, and finally the depart method is executed. Once the agent is deserialized, the receiving server verifies the credentials of the agent. If verified, an acknowledgment is sent to the sender. Otherwise, if an error occurs before the agent can be activated, an exception is returned instead. On a transfer error, the sending server informs the agent by raising an exception that its transfer request failed. On receiving a positive acknowledgment of transfer completion, the sending server updates the agent's location with Ajanta's name service, and cleans up its internal data structures to indicate that the agent is no longer hosted by it. Section 3.7.4 describes the primitives related to Agent communication and synchronization.

3.7.4 Primitives Related to Agent Communication

Agents sometimes need to communicate with each other to accomplish some task. Suitable inter agent communication primitives must therefore be made available. Inter agent communication can be established using different mechanisms. One approach is to provide message-passing primitives, which allows agents to either send asynchronous datagram style messages, or to set up stream based connections to each other. Method invocation is another

approach for communication in object-based systems. If two agent objects are collocated on a server, they can be provided references to each other, using which they invoke operations.

Ajanta allows agents to acquire safe references to collocated agents. For agents that are not collocated, remote method invocation can be provided. Collective communication primitives can be useful in applications that use groups of agents for collaborative tasks. Such primitives can be used to communicate with or within an agent group. Other group coordination mechanisms such as barriers can be built upon these primitives. Communication can also be implemented using shared data. For example, in Ajanta, two or more agents can gain access to a shared object, which can then be used to exchange information.

Another metaphor for agent communication is event signaling. Events are usually implemented as asynchronous messages. An agent may request the system to notify it when certain events of interest occur, such as agent creation, arrivals, departures, check pointing, and so forth. This is referred to as the publish subscribe model of event delivery. Another model is to provide broadcast of events to all agents in a group.

3.7.4.1 Agent-Agent Communication

Agent communication is a major part of the system, as two agents may need to communicate and share information between them to complete some task. Two approaches to agent communication are described later in this section.

Communication Using Resource Proxies: Two agents located on the same server can utilize the proxy based resource access mechanism to communicate among themselves. An agent class can implement the Resource and AccessProtocol interfaces. The agent then registers itself with the server as a resource, and can provide proxies, which allow other

agents to communicate with it. Collocated agents may also communicate via shared access to a resource on their server. However, in many applications, agents residing on different servers may need to communicate or synchronize with each other. Thus a remotely invocable communication mechanism is necessary. Here too the proxy classes must be on the server's classpath. If an agent is allowed to present an RMI interface to the outside world, it opens up a security loophole. The server may grant the agent itself access to certain resources. From the viewpoint of an agent's security, the RMI interface provides a conduit through which unauthorized principals on remote sites may try to interfere with the agent. We need to authenticate incoming connections, so as to control the set of principals, which have indirect access to a resource. The proxy interposition concept is used here to control incoming connections. From the server's security viewpoint, it is necessary to appropriately control an agent's access to the server's communication resources. All incoming RMI invocations are thus intercepted by a proxy object, which passes the RMI call through to the agent object and relays the results back to the caller. The code for the proxy class is loaded from the server's classpath.

Installing a Proxy RMI Server of an Agent: An agent wishing to make itself available for remote invocations by other entities in the system must use the createRMIProxy primitive of the agent environment object. The agent specifies the interface that it intends to support, and requests the server to create and install an RMI proxy. This method executes the following steps. First, it makes sure that the server's security policies allow this agent to install a proxy object for accepting RMI calls. If so, it looks for the corresponding proxy class for that interface on its classpath and creates a proxy instance (containing an embedded reference to the agent object). It then registers the proxy object with the local RMI registry under the agent's name. If the proxy class is not available locally, the createRMIProxy fails—the

agent's code base is not relied upon to provide a safe proxy class. Thus the proxy code is trusted to be safe. When a remote entity wishes to communicate with such an agent, it first finds the current host server for the agent by querying the Ajanta name service. It also obtains the URL for that server's RMI registry. It then queries that RMI registry using the agent's name. The RMI stub returned by the RMI registry however points to the agent's RMI proxy. Section 3.7.5 discusses the issues of the primitives for agent monitoring and control.

3.7.5 Primitives Related to Agent Monitoring and Control

An agent's parent application may need to monitor the agent's status while it executes on a remote host. If exceptions or errors occur during the agent's execution, the application may need to terminate the agent. This involves tracking the current location of the agent and requesting its host server to kill it. Ajanta provides a terminate primitive for this purpose. Similarly, the agent owner may simply recall its agent back to its home site and allow it to continue executing there. This is equivalent to forcing the agent to execute a migrate call to its home site. The owner can use an event mechanism to signal the agent, or raise an exception remotely. The agent's event/exception handler can respond by migrating home. Ajanta provides a retract operation using which, a user can recall their agents from the electronic mall if they run out of digital cash. This capability of remotely terminating and recalling agents raises security issues only an agent's owner should have the authority to terminate it. Thus, some authentication functions need to be built into these primitives, (i.e., the system must ensure that the entity attempting to control the agent is indeed its owner, or has been authorized by the owner to do so). Ajanta is the only system, which performs such authentication. In order to determine whether it needs to recall/abort an agent, the owner must be able to query the agent's status from time to time. Such queries can be answered by the agent's host server, which keeps track of status information for all agents executing on its

site. If the owner needs to make a more application specific query, which can only be answered by the agent, it simply communicates with the agent via the usual agent communication primitives. Table 3.2 gives a summary of Communication and control primitives of various systems. Section 3.7.6 discusses the primitives related to fault tolerance.

3.7.6 Primitives Related to Fault Tolerance

A checkpoint primitive creates a representation of the agent's state, which can be stored in non-volatile memory. If an agent crashes, the owner can initiate a recovery process. It can determine the agent's last known checkpoint, and request the server to restart the agent from that state. In addition to the checkpoints themselves, agent servers can also maintain an audit trail so as to allow the owner to trace the agent's progress along its itinerary, and potentially determine the cause of the crash. If an agent encounters an exception that it cannot handle, its server can take suitable actions to assist the application with recovery. This approach is not supported in Ajanta. Alternatively, the server can simply transfer the agent back to the owner. This allows the owner to inspect the agent's state locally and restart it with appropriately corrected state. Ajanta supports the latter approach. Section 3.7.7 discusses the primitives related to security of the system.

Table 3.2 Communications and Control primitives

System	Communication Primitives	Events and Monitoring	Agent Control
Telescript	Local method invocation after co-location.	Events supported at language-level.	Not Supported.
Tacoma	Agents can co-locate and exchange *briefcases* (i.e., data, using *meet*.)	Not supported.	Not Supported.
Agent Tcl	Message passing using *agent_send and agent_receive*. Stream based communication-using *agent_meet, agent_accept*.	Events are identical to messages.	Not Supported.
Aglets	Send/Receive *Message* objects. Supports synchronous, one-way, future-reply communication modes.	Not supported.	Force agents to return using *retract primitive*. No access control provided.
Voyager	Supports RMI / CORBA / DCOM Synchronous one-way, future and multicast invocations.	JavaBeans compliant event model.	Not Supported.
Concordia	Local method invocation after co-location. Integrates with CORBA. Multicast possible using the *AgentGroup* construct.	*Publish-Subscribe* as well as multicast events.	Not Supported.
Ajanta	Local method invocation via proxy after co-location, RMI via proxy.	Agent status queries supported by servers.	Request agent to return using *recall* primitive. Force immediate return using *retract*. Kill agents using *terminate*. Access control provided.

3.7.7 Primitives Related to Security

Agents may pass through untrusted hosts or networks during their travels, the agent programmer need primitive operations for protecting sensitive data. This includes primitives for encryption and decryption which protect the privacy of data, as well as message sealing or message digests using which any tampering of the code or data can be detected. Digital signatures and signature verification primitives may also be needed to establish authenticated communication channels. If public key cryptography is used, the programmer needs to have a secure key pair generation primitive, as well as a key certification infrastructure. Primitives related to the encoding, allocation and disbursement of digital cash might also be required. An agent's identity, it's certified public key, digital cash allocation, its owner using suitable operations provided by the system can encode constraints on its access rights, and so forth. into its credentials. This primitive is not supported on the Ajanta system.

CHAPTER 4
MOBILITY AND COMMUNICATION IN MOBILE AGENT SYSTEMS

4.1 Introduction

In this chapter the author gives an overview of the state-of-the-art in agent communication systems, the author decided to evaluate Java-based and non-Java-based agent platforms, various evaluation criterion like, the point at which execution is restarted at a defined entry point or from the beginning will be evaluated. The following Java-based platforms were selected for evaluation:

The two following non-Java-based were chosen for evaluation:

- April from Imperial College [39,48]
- D'Agents from Dartmouth College [40,41]

The four following Java-based platforms were chosen for evaluation:

- Aglets Software Development Kit from IBM [43,46]
- Grasshopper from IKV++ [42,45]
- Odyssey from General Magic [49]
- Voyager from ObjectSpace [44]

The author will first analyze the non-Java based systems and then the Java based mobile agent platforms. The communication services were analyzed and a comparative table is provided, the various limitations to the currently existing protocols have been pointed out, chapter 6 provides a solution by which some of the limitations of the current mechanisms can be overcome.

4.2 April 3.0

April (Agent Process Interaction Language) is a process-oriented language [39] designed for implementing intelligent applications on a network. April is also the name of corresponding execution platform. The April system is written in C and the agents are programmed in the April language. The agent transport and communication is based on a proprietary protocol on top of TCP/IP.

4.2.1 Agent Mobility

April treats functions and procedures as first-class data values[1], which can be any size from code fragments to whole agents [39]. Mobility is achieved by sending these values in messages to remote processes, which can then either incorporate them into their own code (for methods) or fork them as new processes (for agents). The execution stack cannot be serialized**.** The execution is restarted at a defined entry point [39].

Serialization is performed transparently to an internal format and transport uses the same mechanism as ordinary communication. As the entire code of the agent can be serialized, a class loader is not used [39].

Since the original agent remains in place after the message is sent, all previous communication links remain intact. The original agent can either continue as before, exit, or become a proxy that forwards messages to the new agent. If it exits, but had a publicly named handle, new incoming messages will be held by the communication server for later delivery to allow services to be stopped and restarted without disruption. However, if it had a generated handle, new incoming messages will be discarded.

[1]A language supports some data type as *first-class* when the objects of the type can be created and used as data at run time. First-class data can be kept in variables, and passed to and returned from functions. In dynamically typed languages, first-class data can also have its type examined at run-time.

Agents explicitly decide for themselves which destination to travel to next. There is no specific itinerary mechanism. No distinction is made between stationary and mobile agents.

4.2.2 Agent Communication

April processes communicate with each other by special message *send* and *receive* commands defined within the language. It makes no difference whether the receiving process is local or remote. April agents can transparently send messages to other agents located anywhere on the network, as long as their handles are known.

The base implementation for the message-passing operation is TCP/IP between April communication servers running at a well-known port number on each machine. These servers then deliver messages to the intended process. Hence a separate port for each process is not required.

Messages can contain any April data type, ranging from atomic types such as symbols or integers to tuples, lists, and higher-order types such as function and procedure + structures such as KQML [40] messages to be sent. Constructors are not necessary. The expressions to be sent is given as an argument to the message send command. Strong typing[2] is not used—any message type can be dynamically created and sent to any agent. If the receiver is not capable of handling the message, it can choose to ignore it.

All incoming messages are stored in a single queue. Messages originating from different senders may be interleaved, but all messages from the same sender are guaranteed to arrive in order [40]. A process does not have to process its incoming messages in strict

[2]Implies that values of one type cannot be assigned to variables of another type.

order, as it can scan the entire queue first looking for messages matching a particular pattern, (e.g., priority messages).

The base communication mode of April is one-way asynchronous. No event is generated when a message arrives. The receiver must explicitly perform a (blocking) message receive. Blocking the sender after each message and requiring an immediate acknowledgement to be sent by the receiver can implement synchronous communication. Two-way asynchronous communication can be implemented by having the sender stop at some point and block waiting for a reply. Multicast messages can be sent to any list of processes.

Messages are completely location-transparent and can be stored for later delivery to a process that has not yet started, or forwarded by a chain of proxies (see 1.2.1 Agent Mobility) an arbitrary number of times to follow a process that has migrated. Some processes may also act as blackboards, forwarding incoming messages to other processes which have registered interest in them by placing patterns on the blackboard. Any messages matching these patterns will then be forwarded.

There is no built-in directory support. Agents can only discover handles by using pre-specified public names, by receiving them in messages, or by forking children.

4.3 Aglets Software Developer Kit 1.0

The ASDK is entirely written in Java. It requires the JDK 1.1 or higher to be installed. The Aglets transport and communication is based on the proprietary Agent Transfer Protocol (ATP), which is modeled on the HTTP protocol [43]. It is accessed through the Agent Transfer and Communication Interface (ATCI) [45], which allow an abstraction from the underlying transport protocol.

4.3.1 Agent Mobility

Since ASDK is implemented in Java, the serialization of the execution stack is not supported. Therefore one fixed entry point is defined which is called at every arrival. This limitation requires that the current execution state of the aglet task has to be stored by itself in a serializable Java data structure. Then the execution state can be restored after migration from that data. ASDK uses the Java-Object-Serialization of the JDK to marshal and un-marshal Java-objects. Aglets can migrate to another context by calling the *dispatch* method with the URL of the remote host as the argument. The aglets and the contained data objects will be serialized. The classes necessary to build the aglet in the new context are loaded by the class loader from the given code base [45].

Currently, the proxy-objects do not keep track of roaming aglets. Once an aglet is dispatched, the proxy-objects referring to it are no longer valid. A mechanism for preserving the reference to the moving aglet is announced for the future [45]. A migrating proxy-object remains valid.

An aglet can migrate on its own will, forced by another aglet, by the user via the GUI, and by the agent system. It can also be fetched back by using the *retract* method from the GUI. Additionally, the ASDK provides the itinerary-class, which can be used to specify elaborate travel plans. The ASDK does not distinguish between mobile and stationary agents.

4.3.2 Agent Communication

Aglets communicate with each other by message passing [43]. That allows dynamic communication with unknown as well as with well-known aglets. The communication is based on a simple call back scheme that requires an aglet to implement the necessary

handlers for each kind of message. Invoking a *send-message* method on the proxy-object sends the messages. The proxy-objects enable location transparency for aglets with the restriction that proxy-objects for moving aglets become invalid. For the communication it makes no difference whether the receiving aglet exits locally or remotely [43]. The generation of proxy-classes is not required because all aglets have to implement the same message handler.

Messages may contain arguments of any Java type that implements java.io.Serializable [45]. Using predefined constructors can create atomic messages. Non-atomic messages with multiple arguments are handled as a list of key and value pairs. A remote message does not cause any transfer of byte code. The classes used in the message have to be installed in both hosts.

ASDK supports synchronous, asynchronous and multicast messages. Currently, one-way communication is documented but not implemented [43]. For asynchronous messaging an object reference is returned which can be used to check for incoming replies. Multicast messages can be sent to all aglets within a local context that have subscribed to this kind of message.

All incoming messages are stored in a message queue and then handled one by one. This serializes the message handling and ensures that the handling of the current message is finished before the message handler of the aglet is invoked again. This process can be customized to allow the handling of the messages in order of their priority depending on their type and to synchronize messages with the aglets' state.

The ASDK defines a number of methods that are called in case of a major event (e.g. on creation, on cloning, on arrival). The aglets programmer to react to a special event can

overwrite these methods. Message forwarding and dynamic invocation interfaces are not supported [43].

The ASDK provides a very simple directory services for aglets. A list of hosted aglets can be retrieved only for the current context. Since proxy-objects are not valid after the migration of the referenced aglet, it is very difficult to keep track of a moving aglet and to establish a connection to a remote aglet with unknown Globally Unique IDentifier (GUID). For the creation of proxy-objects the URL of the host system and the GUID is needed.

4.4 D'Agents 2.0

The Agent TCL platform is a modified version of the Tool Command Language interpreter (TCL), which aims at providing a basic support for mobility. This support is achieved via a modification of the TCL kernel which offers a command to freeze the execution of a TCL script, catch the execution state, move it to another computer, and then resume the execution at the next instruction of the script.

The platform has just been renamed D'Agents as its last version is henceforth designed to take into account other programming languages, not just TCL. Today, D'Agents is the only available target language, but the integration of Java, Scheme and Python is on the way [40,41].

D'Agents 2.0 (with TCL) comes with a basic architecture and a basic command set for communication and migration. It also offers security enforcement, resource access control, and agent life cycle management. The platform is written in C and the agents are written in TCL. For transport and communication purposes D'Agents uses a proprietary protocol based on TCP/IP or on e-mail [40].

4.4.1 Agent Mobility

An agent controls its own mobility by using the unique *agent_jump* command. The modification of the TCL interpreter allows the capture of the entire execution stack of an agent. On arriving, the execution is resumed at the command following the migration instruction. When the agent is restarted on its new host, it has a new local identifier and its name is not registered anymore. All the pending messages are lost, and the direct connections (see the communication section) are merely broken. Since the global identifier always includes location information, any reference to the moving agent is obsolete. From a general point of view, every local resource (handlers to files, sockets...) of the origin site is de-allocated.

There are no predefined structures allowing the representation of the itinerary of an agent though it would be quite straightforward to implement [40], as TCL makes it easy to handle lists of destination-site or command-to-launch tuples. Nevertheless, such a high-level representation is not useful as the execution state is saved by the *migration* command.

There is no concept of stationary agent, but it is possible to specify that an agent cannot move, so that it is forced to be stationary.

4.4.2 Agent Communication

Two communication modes are available: Messages and Direct Connections. The former, a mail service, is managed by the agent daemons on the sender and recipient sites. Two commands allow agents to send and receive asynchronous bytes messages, which may hold arbitrary data or TCL scripts. The latter consists of a message stream between two agents (typically a socket connection). It is established after a synchronous request from one agent and an acceptance from the target agent. By accepting a direct connection, an agent gets a privileged communication channel with another agent to exchange information

without utilizing the mail service. In any case, its host machine and its local identifier specify the target agent of a communication. The local identifier may be a name registered in the local daemon, which offers a local directory service.

4.5 Grasshopper 1.1

Grasshopper is the first available mobile agent platform, which is compliant to the MASIF standard. It is being continuously developed since 1997 by the IKV++. Grasshopper 1.0 was released in March 1998 for UNIX and Window NT systems.

4.5.1 Agent Mobility

Java does not provide mechanisms for capturing the execution stack. Therefore an entry point can be specified in the *move* command, which is called after the arrival of the agent, or the default *entry* method is called.

In Grasshopper all proxy-objects keep valid even if the referenced agent or the proxy-object itself moves [42]. If the referenced agent has left its location, the proxy-object is updated with the new location information fetched from the region registry.

An agent can migrate on its own will, forced by another agent, by the agent system or the user via the GUI. Grasshopper does not specify any special itinerary pattern.

There are two different kinds of agents in Grasshopper, stationary agents and mobile agents. A stationary agent executes on the agent system where it was started. It cannot migrate by itself, but it can be moved by external entities. Stationary agents can be used to offer services, which are related to their agency. They improve the core functionality of an agency.

4.5.2 Agent Communication

Inter-agent communication is based on method invocation [45]. In order to communicate with another agent, its GUID and the corresponding proxy-class are required. The GUID can be found in the region registry. The proxy-class has to be generated before the agent is compiled using the stub generator of the Grasshopper package. If the proxy-class is not available, a dynamic invocation interface can be used. Optionally it is possible to specify the location of the target object in a communication request. Grasshopper supports location transparent communication. Synchronous, asynchronous and multicast communication is possible. Grasshopper provides callbacks on important events, e.g. if a message arrives.

The region registry of Grasshopper can also be used to find the GUID of communication partners. Theoretically the region could have a global scope [42].

4.6 Odyssey 1.0 Beta 2

Odyssey is General Magic's mobile agent system. It is implemented as a set of class libraries that provide support for developing distributed, mobile applications.

The libraries allow the building of *places* in the network as needed for agents to travel between. Odyssey is implemented in Java and requires the JDK1.1. It is transport independent by the use of a transport API. Odyssey provides implementations of this API for RMI, IIOP, and DCOM. In order to use IIOP as transport and communication mechanism, VisiBroker [13] for Java by Visigenic (Inprise) is required.

4.6.1 Agent Mobility

Java does not provide a way to capture the execution state of a Java thread. Thus, when an Odyssey agent is transported from one system to another, the agent's thread is

restarted at the destination at a predefined point [49]. Therefore Odyssey provides the worker class, which is a subclass of the Odyssey agent class. A worker is structured as a set of tasks and a set of destinations. At each destination, the worker executes to completion the next task on its task list. An Odyssey worker may manipulate its task list at any point during its travels [49].

Odyssey uses the Java-object-serialization to transfer the agents. Since Odyssey does not support remote communication, all object references become invalid during a migration.

4.6.2 Agent Communication

Documentation concerning the agent communication in Odyssey is nearly non-existent. Odyssey communication is based on agents meeting in the same place. Remote communication is not possible [49]. By the use of the function *meet*, agents can find other agents in the place and exchange object references, which can be used for further communication. Other mechanisms are not supported.

A finder interface for places and agent systems is define in the API. The directory service for agents seems to be limited to the local host.

4.7 Voyager 2.0 Beta 1

Voyager is a Java-based and agent-enhanced Object Request Broker. It is entirely programmed in Java. It requires the JDK 1.1 or higher to be installed.

The agent transport and communication is based on a proprietary ORB on top of TCP/IP. Since version 2 support for CORBA communication in included in the ORB.

4.7.1 Agent Mobility

The serialization of the execution state is not possible in Java. Voyager provides advance possibilities for defining the entry point after migration [44]. When objects move, they leave behind *forwarders* to forward messages and future communication requests. All virtual object references keep valid, even if the objects move. When the target object cannot be located, the forwarders are used to find the new location of the object. The return value is tagged with the remote object's new location so that the virtual reference can be updated. Subsequent messages and request are sent directly to the new location of the remote objects. An agent in Voyager can move on its own will or can be forced by other objects.

Voyager provides a simple itinerary mechanism [44] to define a collection of tasks to be performed at a sequence of locations. The locations to be visited can be stored as a Vector of addresses, and an agent can move easily from location to location, performing tasks as it travels. The locations can consist of an arbitrary combination of target objects and programs. Voyager uses regular Java features to execute an itinerary.

Voyager does not explicitly use the concept of stationary agents [44].

4.7.2 Agent Communication

In contrast to other ORBs, Voyager uses lightweight agents called messengers to deliver messages. The communication to remote objects is transparent. Since version 2.0 beta 2, the need to pre process the class files to generate proxy-objects is eliminated [44]. The proxy-classes are dynamically generated at runtime. Voyager supports the following types of communication:

- Synchronous messages.
- One way messages.

- One way multicast messages with subscribe mechanism.
- Asynchronous or future messages.
- Events.
- Dynamic invocation.

Voyager provides 'Space–Scalable Group Communication' to send broadcast messages or events to a great number of objects [45]. In order to efficiently distribute the messages, clusters of objects in the target group are stored in local groups called subspaces. These Subspaces are linked to form a larger logical group called a Space. When a message is sent into one of the subspaces, it is cloned to each of the neighboring subspaces before being delivered to every object in the local subspace. Voyager implements a special mechanism in each subspace, which ensures that no message is processed more than once. This architecture allows a parallel distribution of the messages to every object in the Space and enables a good scalability of the system.

Voyager provides a comprehensive event and call back service. Messages are forwarded by the use of the forwarders left behind by migrating objects. Voyager allows the use of dynamic invocation interfaces and provides a global directory service for remote object look-up. This service is built of distributed directory services, which are linked together to form a single, federated directory service [45].

4.8 Platform Comparison

Here the author presents a comparison of all the mobile-agent platforms evaluated for their mobility and communication.

4.8.1 Agent Mobility

The Serialization of the execution stack is not possible in all agent platforms based on Java and in April. Hence, fixed entry points are defined at which the execution of the agent is resumed after migration. Grasshopper and Voyager allow specifying these entry points by the agents. Only D'Agent is able to transport agents including all execution stack information. That may simplify the programming of agents because their state must not be stored in data structures before migration. The missing serialization of the execution stack is only a minor drawback, which can be overcome by an adapted programming scheme. Therefore ASDK, Odyssey and Voyager provide special itinerary mechanisms for common travel patterns.

A major problem of mobile agents is the validity of object references. Only Voyager and Grasshopper provide platform mechanisms to automatically update the references to moving agents. In all other platforms this task is left to the agent programmer, which results in overhead in each agent.

The agents, corresponding to the concept of mobile agents, in any case control the migration. Additionally, ASDK, Grasshopper and Voyager allow the control of migration by other agents and applications. ASDK and Grasshopper allow moving an agent by the use of the GUI.

In contrast to all other platforms, Grasshopper distinguishes between stationary and mobile agents according to the MASIF specification [13]. A stationary agent is not able to migrate at its own will, but it can be forced to do so by another entity. Stationary agents are usually used to enhance the core functionality of a Grasshopper agency.

4.8.2 Agent Communication

Table 4.1 lists the following features of agent communication.

- **Communication Type**: Describes the basic communication mechanism.
- **Proxy-class Generation Required:** Is the pre-processing of class files to generate the proxy-classes required?
- **Remote Communication:** The general ability of communication with remote objects.
- **Location Transparency:** This denotes the fact that in a communication the location of the communication partners is transparent and is internally handled by the underlying communication mechanisms.
- **Synchronous Communication:** The way of communicating synchronously, i.e. the sender blocks until the receiver returns.
- **Asynchronous Communication:** The way of communicating asynchronously, (i.e., the sender continues its execution), regardless of the message receiver. The result of the method invocation can be returned later.
- **One-way Communication:** A special case of asynchronous communication, where the sender just sends a message, without expecting a result.
- **Multicast Communication:** The ability of a sender communicating with multiple receivers.
- **Event Services/Call Back:** The feature of sending and handling events asynchronously.
- **Message forwarding:** The ability of forwarding a message to the new position of the receiver.

- **Dynamic Invocation Interface:** Coined by the CORBA standard, this term denotes the ability of invoking a method of another agent when the proxy is absent.

Table 4.1 A comparative study of different MA platforms.

Criterion	April	ASDK	D'Agents	Grasshopper	Odyssey	Voyager
Communication type	Message passing	Message passing	Message passing	Method Invocation	Method Invocation	Method Invocation
Remote Communication	Yes	Yes	Yes	Yes	No	Yes
Proxy-class Generation Required	-	-	-	Yes	-	No
Location Transparency	Yes	Almost transparent	Almost transparent	Yes	-	Yes
Synchronous Communication	Yes	Yes	No	Yes	Yes	Yes
Asynchronous Communication	Yes	Yes	Yes	Yes	No	Yes
One-way Communication	Yes	No	Yes	No	No	Yes
Multicast Communication	Yes	Yes	No	Yes	No	Yes
Dynamic Invocation Interface	-	No	-	Yes	No	Yes

In contrast to all other platforms Odyssey does not support remote communication. This may be due to the fact that was not developed any further in the last year. Since

Odyssey does not provide remote communication, Grasshopper is the only platform, which requires the generation of proxy-classes. In Voyager the proxy-classes are generated at runtime.

In April, ASDK and D'Agent the communication is location transparent as long as the agents do not move. Otherwise a new communication has to be established. Grasshopper and Voyager allow for completely location transparent communication. The references to moving agents are automatically updated by the agent system.

April, ASDK, Grasshopper and Voyager support multicast communication. Voyager uses a specialized architecture with super-spaces and subspaces to efficiently deliver the messages.

ASDK and Grasshopper provides a simple call back service on important events. Voyager has implemented a complex distributed event services.

No evaluated platform has currently specified the content or the semantics of messages or has implemented an agent communication language.

CHAPTER 5
CORBA AND MOBILE AGENT TECHNOLOGY

5.1 Introduction

In this chapter the author gives an overview as to how Mobile agents (MAs) can make use of already defined CORBA objects and services and vice versa offer their services to other CORBA objects.

MAs are considered today as one of the key middleware technology for the implementation of flexible, nomadic applications residing within service and/or network layers.

In contrast to the early days of this technology, today there exists a common understanding that Mobile Agent Technology (MAT) can be regarded as an enhancement of Distributed Object Technology (DOT), such as the Object Management Group's Common Object Request Broker Architecture (CORBA) [31].

The first step in that direction is achieved by approving the Mobile Agent System Interoperability Facility (MASIF) specification within the OMG [32]. This specification follows the architectural guideline as proposed in the OMG's Object Management Architecture in which higher level functional capabilities are specified in so called Common Facilities, which are based on lower level functional capabilities provided by CORBA services.

The MASIF specification's target is to enable a minimum interoperability among different agent systems through the definition of two key interfaces. However, it does not yet cover a mapping of the key capabilities of an agent system's services to CORBA services. The basic assumption is that object mobility will be a fundamental capability of

CORBA [31,32], which then allows a more flexible, location aware service provisioning within the field of terminal and personal mobility. This means that in the midterm the capabilities provided by today's mobile agent platforms, which provide that object mobility in form of add-on services on top of CORBA, can be incorporated into CORBA directly, (i.e., the mobility of CORBA objects) is supported by the core and an appropriate set of CORBA services, such as (the Life Cycle Service, Persistency Service, Trader Service, etc.). The key questions in this context are:

- Are the already specified CORBA services (i.e. their interfaces) sufficient to provide the required object mobility capabilities?
- If yes, how could object mobility be provided?
- If not, what additional interfaces or object services are required?

This chapter describes the requirements for supporting object mobility in CORBA based on the capabilities provided by state of the art mobile agent platforms.

5.2 CORBA Common Object Services

The current Common Object Services Specification (COSS) [33] defines the following services:

- Concurrency Control Service
- Event Service
- Externalization Service
- Licensing Service
- Life Cycle Service
- Naming Service
- Object Collection Service
- Persistent Object Service

- Property Service
- Query Service
- Relationship Service
- Security Service
- Trading Object Service
- Transaction Service
- Time Service

The Objects by Value Service is also considered. The following paragraphs describe briefly those services, which are of special interest for the integration of CORBA and mobile agent technology with respect to the mandatory MA platform requirements listed in section 3.

5.2.1 Externalization Service

The Externalization Service defines protocols and conventions for externalizing and internalizing objects [33].

Externalizing an object means to store the object's state in a stream of data, (e.g., in memory, on a disk drive, or across the network). An externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. Internalization means the recovery of an externalized object in the same or a different process.

5.2.2 Life Cycle Service

The Life Cycle Service defines conventions for creating, deleting, copying, and moving objects [33]. Because CORBA-based environments support distributed objects, the Life Cycle Service defines mechanisms that allow clients to perform life cycle operations on

objects in different locations. Object creation is defined in terms of factory objects, where a factory is an object that creates another object.

5.2.3 Naming Service

The Naming Service provides the ability to bind a name to an object relative to a naming context [33]. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name means to determine the object associated with the name in a given context.

5.2.4 Trading Object Service

The Trading Object Service [33] facilitates the offering and the discovery of service instances of particular types trader, (i.e., an object that supports the Trading Object Service), maintains information about services, (i.e., the service type, the service's object reference, and various service-specific properties). A client may contact a trader in order to export services, (i.e., to register new services in a trader), or to import services, (i.e., to search for specific services). Several traders can be connected with each other, thus increasing the available search scope for requesting clients.

5.2.5 Security Service

The CORBA Security Service specifies the functionality to authorize principals, control their access to objects, and secure the communication among objects. Once the functionality of this service is provided, clients and servers, both acting in behalf of principals, are sure that a trustworthy ORB environment is provided for communication purposes [33].

5.2.6 Objects by Value

The Objects by Value Service provides the facility to pass on object by value rather than by reference as the traditional CORBA approach. This service respects any constrains of the current CORBA and is fully incorporated into the CORBA model [31,34].

By using Objects by Value, an application can explicitly make a copy of an object and replicate it onto another physical node. By definition, the copied object does not have any relationship with the original one.

5.3 Key Capabilities of Mobile Agent Platforms

Mobile agents need a sophisticated runtime environment that supports their entire life cycle, including among others their creation, termination, and migration. Besides, users must be able to manage, (i.e., to monitor and control their agents).

The following paragraphs describe the basic capabilities that are provided by a mobile agent platform.

Management Support: It is necessary for agent administrators to be able to monitor and control their agents. The control aspect comprises at least the creation and termination of agents. Monitoring of agent's means in the first place their localization in a distributed environment. Due to the agents' mobility and active behavior, this is not a trivial task.

Mobility Support: A special mobility support must be provided by the platform, supporting both remote execution and migration. The mobility supports the transfer of the state and the classes representing an agent.

Support for Unique Identification: Mobile agents must be uniquely identifiable in the whole distributed environment. This is the inevitable precondition for accessing a specific agent.

Apart from the mandatory capabilities above, the following optional platform capabilities should be considered:

Transaction Support: An important aspect is the support of correct and reliable execution of agents in presence of concurrency and the occurrence of failures. Thus, transaction support is desirable.

Security Support: Important aspects are authentication, (i.e., the determination of an agent's or agent system's identity), and access control of resources or services. Besides, in order to guarantee privacy and integrity, important information such as code and internal data of a migrating agent, should be encrypted before the transfer.

Communication Support: Agents should be able to communicate with each other as well as with the surrounding environment.

5.4 Mobile CORBA Agent

A mobile CORBA agent – or short just named as agent - is a group of CORBA objects that provides multiple CORBA interfaces [35,36]. An instance of an agent has the ability to migrate from one physical network location to another without loosing its internal state, (i.e., the states of all objects associated with that agent). By definition, all CORBA objects associated with an agent can only be migrated as a whole [31].

In order to support the mobility, an agent is always associated with two CORBA objects, which one realizes the LifeCycleObject interface [37] as defined in the Life Cycle Service specification and the other the Streamable interface as defined in the Externalization Service specification [32].

An agent can be created either from templates (classes) or by cloning an existing instance of an agent. Agents are delivered as package consisting of the class and possibly a

serialized version of an agent instance itself. Unpacking can cause the serialized agent to reinstate itself at the destination [32].

The additional CORBA objects, (i.e., consequently CORBA interfaces), associated with it, characterize the provided service of an agent. The following figure sums up the content of these Sections.

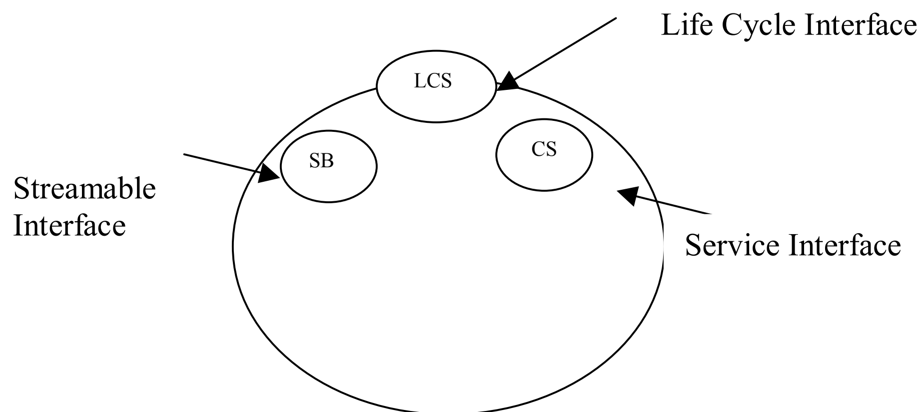

Figure 5.1 A mobile CORBA object.

5.4.1 Structure of a Mobile CORBA Agent

The basic structure of an agent is relatively simple. There are two kinds of object types, a core (stateful) object type and a bridge (stateless) object type [37].

5.4.1.1 Core Object Type

A core object type contains the state of an agent. An essential property of this object is that its implementation is always local, in a particular programming language, (e.g., Java). This object has no identity and is not registered with the ORB. The realization of the provided service of agent is contained within this object. This object in order to access the

service, local interfaces (API) is supported. Such a core object is composed of native objects.

Additionally, the programming language used for implementing the agent must support the serialization of objects, (i.e., write to and read from a stream). The serialization of objects is not part of CORBA [31]; it is matter of the programming language. The core object is the part that is involved in the migration process.

5.4.1.2 Bridge Object Type

A bridge object is a CORBA object that inherits from *CORBA::Objects* and supports normal object reference semantics. The interface associated with this object is realized as an inter-working bridge between the other CORBA objects, acting as clients, and the core object of an agent, acting in the server role. Thus, a bridge object translates incoming CORBA requests and makes the appropriate local interface (API) calls provided by the core object.

5.4.2 Identification of a Mobile CORBA Agent

Input for this Section will be provided, once some clarifications of which part of an agent has to be identified is made.

The agent identification is still a heated debate. Traditionally, each CORBA object that realizes an interface is registered with the ORB, and as specified in the CORBA specification [31], a unique Interoperable Reference (IOR) is associated with each CORBA instance. Since an agent support multiple CORBA interfaces means that there are going to be at least three IORs, which are associated to the Life Cycle, Streamable, and Service Interface. There are few questions that have to be answered:

- Should all CORBA objects be registered themselves to the Naming Service by their names? But, then the group relationship among these CORBA objects is not expressed [31].
- Should an agent only register a well-defined CORBA object, which provides an interface for accessing the interfaces of the other CORBA objects, associated with the agent? That means that the other interfaces are local and are therefore not directly resolvable.
- Is an agent a CORBA object with an object reference or just a logical construct to group other CORBA objects?
- Should a new CORBA service be identified in order to create unique name or identifier for agents?
- Should an agent be identified by a naming graph? For example, a CORBA object's compound name is stated: <name of the agent, name of the CORBA object associated with the agent>, where the first part is used to name the agent context and the last component denotes the bound object. This approach has the advantage that the group relationship is expressed within the naming graph.

5.5 Mapping of CORBA Services to MAP Capabilities

This Section explains how the COSS services introduced in Section 2 can be used to handle mobile CORBA agents instead of usual, static CORBA objects.

The main difficulties are the following:

- Mobile CORBA agents are able to migrate from one physical network location to another. The ORB must be able to handle this mobility aspect, for instance by managing the availability of the access part to the agent, which changes its location

after each migration. However, please refer to Section 5.1 for more information about the mobility aspect in a CORBA environment.

- During migration, an agent's state and code must be transferred to the new location. The environment to handle this task must provide mechanisms.
- The CORBA environment must handle a synchronization process between the creation and deletion of the instance at the new and original location.
- The instantiation of an object requires a suitable runtime context, which depends on the language in which the object is implemented. For instance, a Java object can only be created inside a Java program that is running on top of a Java Virtual Machine (JVM). The CORBA environment must maintain suitable runtime contexts for all supported implementation languages and further agent-related requirements. This aspect is discussed in Section 6.

5.5.1 Initial Thoughts about Mobility

The migration of mobile agents can be split into the following procedures:

1. Externalization of the agent's execution state
2. Pack and transfer of the externalized state and the code to the destination location
3. Reinstatement of a new agent instance at the destination location
4. Internalization of the agent's execution state
5. Removal of the original instance at the source location

The interaction steps 3 and 5 show that there exist two instances of an agent during the migration process. The new instance at the new location is not identical with the one at the original location. After the removal of the instance at the original location, those CORBA objects associated with the agent can register themselves to the ORB at the new location and therefore will get new IORs associated to them. In the specification of the

Generic Inter-ORB Protocol (GIOP) [31], an approach for handling this IOR modification is given:

An ORB must provide so-called "agents" which are responsible for handling client requests at the server side of a connection. Note that these "agents" have nothing in common with the mobile CORBA agents in the context of this chapter. To illustrate the difference, the "agents" specified in the GIOP description are surrounded by quotation marks.

An "agent" has the following possibilities to handle client requests if the IOR specified by the client is not associated with an active object (for instance if the corresponding object has moved to another location and has thus obtained a new IOR):

1. The “agent” knows the new IOR of the object and forwards it to the new address. The result of the request is sent from the object back to the “agent” which forwards it back to the client. This is achieved transparently to the client, which means that the client is not aware of the forwarding procedure and of the new server side location.
2. The "agent" knows the new location of the object, but is not able to forward the request to the object. Instead, the "agent" delivers the new IOR to the client, which is then able to establish a new connection to the object at the new location.
3. The "agent" is able to handle the client request even without contacting the actual object. This may be possible if the "agent" maintains (parts of) the object’s implementation.
4. The "agent" is not aware of the object’s new IOR and returns an exception to the client, indicating that the object is not any more reachable. In this case, the client may retrieve the current IOR of the object by contacting the Naming Service or a trader. For this purpose it is necessary that the object must have modified its entries in the Naming Service or the trader, respectively.

5.5.2 MA Specific Requirements

In Section 2, a general, brief description of those COSS services has been given which are essential for the support of mobile agents. However, the description was not related to mobile agents, but to CORBA objects in general. This section explains the agent-specific aspects and requirements regarding the COSS services.

5.5.2.1 Externalization Service

Agent migration relies on the agent's or agent system's capability for externalization. This means, it must be possible to maintain an agent's object structure, including all fundamental internal information, by means of a stream. The stream can be transferred across the network, and the agent's object structure can be recovered at the destination location.

The Externalization Service defines interfaces for this process, (i.e., for the externalization and internalization of objects). Thus, this service is essential for the support of agent migration.

5.5.2.2 Life Cycle Service

The Life Cycle Service defines methods to create, remove, copy, and move objects. While the methods create and remove are essential for each object, the move method is especially of interest in the context of mobile agents.

This method may be invoked by the agent itself in order to migrate actively, or even by an external client in order to send the agent to another location.

5.5.2.3 Trading Object Service

Due to the agents' mobility, their management is not a trivial task. A fundamental requirement regarding agent management is the ability to locate each agent at any time,

independent of its current location. The Trading Object Service seems to be suitable to support this in a sufficient way. Each agent could be registered within a trader. The Trading Object Service offers the possibility to store any desired property of a registered service. An agent-specific property could be the agent's current location, (e.g., in form of a URL). In this way, a client, (e.g., an agent manager), would be able to track specific agents by specifying their name and retrieving their current location from the trader. Also the trader, such as the identifier or name of the agent owner, can maintain any other agent property. In this way, an administrator would be able to get information about all agents belonging to a specific person or organization, or the owner herself would be able to get the current locations of all owned agents, just by initiating a single request to the trader.

Another important aspect regarding the Trading Object Service in an agent-enhanced CORBA environment is the maintenance of agent systems. Each agent or at least each agent type requires an individual runtime environment (i.e. agent system) in which it can be executed. Before migrating to a new location, the agent must locate an agent system at its desired destination, which is able to support its execution. the proposed solution for this problem is to register all running agent systems in a trader. One essential property to be contained in each agent system entry is a list of agents or agent types are supported by the agent system.

5.5.3 Preconditions for a Mobile CORBA Agent

This section explains the preconditions that must be fulfilled by each agent in order to be executable and manageable in a CORBA environment.

5.5.3.1 Life Cycle Support

As stated in Section 4, each mobile CORBA agent must provide the interface Lifecycle Object as defined in the Life Cycle Service Specification [32]. This interface comprises operations to copy, move, and remove the object/agent.

5.5.3.2 Externalization Support

A fundamental requirement for agent migration is that the agent is externalized, (i.e. important internal information), (e.g., the agent's execution state), is written out into a stream. The counterpart, (i.e., the internalization), means that the state is read out from the stream for the purpose of reinstatement of the agent.

Due to the CORBA Externalization Service specification, an object that shall be externalizable must support the Streamable interface [33].

5.5.4 Mobile CORBA Agent Related Interaction Scenarios

This section shows how the creation, removal, migration, and localization of mobile agents can be achieved just by using COSS services. The proposed solutions may not be very efficient. However, they do not need any modifications or enhancements to the services.

5.5.4.1 Agent Creation

Mandatory CORBA Common Services:

- Life Cycle Service. Required interfaces: FactoryFinder (FF), Factory (F)
- Naming Service. Required interfaces: NamingContext (NC)

Optional CORBA Common Services:

- Trading Object Service. Required interface: Registration (R)

Interaction Scenario:

1. *resolve*: The client looks for a factory finder by contacting the Naming Service.
2. *find_factories:* The client contacts the factory finder in order to find a specific factory that is able to create the demanded agent.
3. *create:* The client invokes the create method of a suitable factory.
4. The factory creates the core object of an agent. Since the core object contains the core implementation, it then creates all the bride objects associated with it.
5. *bind*: The new agent creates a context in the Naming Service and binds all bridge objects to this context. Clients can then resolve the IORs by resolving the component names.
6. *export*: The agent creates an entry in a trader by using the trader's Register interface. This entry contains all-important information about the agent, such as its current location and all offered services.

Note: Several additional steps have to be performed, (e.g., in order to generate a CORBA name), but these steps are not mentioned here for the sake of simplicity.

5.5.4.2 Agent Termination

Mandatory CORBA Common Services:

- Life Cycle Service. Required interfaces: LifeCycleObject (LCO)
- Naming Service. Required interfaces: NamingContext (NC)

Optional CORBA Common Services:

- Trading Object Service : Registration (R)

Interaction Scenario

1. *resolve:* The client looks for the agent to be removed by contacting the Naming Service and resolves the name of the CORBA object that provides the Life Cycle interface
2. *remove*: The client invokes the remove operation on the agent's LifeCycleObject interface. This request is passed to the core object which first deletes all bridge objects.
3. unbind: Next, the agent unbinds all CORBA objects from the agent's context in the Naming Service and destroys its context.
4. *withdraw:* The agent removes its entry in the trader by using the trader's Register interface and finally removes itself.

5.5.4.3 Agent Migration

Mandatory CORBA Common Services:

Life Cycle Service. Required interfaces: LifeCycleObject (LCO), FactoryFinder (FF), Factory (F)

- Externalization Service. Required interfaces: StreamFactory (SF), Stream (S), StreamIO (SIO), Streamable (SB)
- Naming Service. Required interfaces: NamingContext (NC)

Optional CORBA Common Services:

- Trading Object Service : Registration (R)

Interaction Scenario

1. *move*: The client invokes the agent's move method, provided by the LCO interface. Note that this step is usually performed by the agent itself.

2. *find_factories*: The agent looks for a stream factory that is able to create a Stream object for the agent's externalization.
3. *create*: The agent creates a Stream object for its externalization.
4. The stream factory creates the Stream object.
5. *externalize* The agent initiates its own externalization. Note that in [32] this operation is performed by the client.
6. *externalize_to_stream* The Stream object has prepared for storing the streamed agent and orders the agent to write its externalized form. This request -at the CORBA level- is translated to native serialization methods of the core object. The core object serializes itself into native stream forms and maps them to CORBA date types.
7. *write_<data>* The agent sends its externalized form to the Stream object, using various write methods, depending on the data type to be written.
8. *find_factories*: The agent looks for a factory that is able to create a new instance of the agent at the desired new location.
9. *create:* The agent creates a new instance of itself at the new location.
10. The factory creates the agent.
11. *internalize:* The new agent instance contacts the Stream object that maintains the externalization data in order to initiate the internalization process.
12. *internalize_from_stream*: The Stream object has prepared for recovering the streamed agent and orders the agent to read its externalized form.
13. *read_<data>:* The agent reads its externalized form from the Stream object, using various read methods, depending on the data type to be read. Note that, for the sake of simplicity, this scenario assumes that the agent is a single object.

14. *rebind:* The agent is now internalized and ready to continue its task processing at the point where it was interrupted before the migration. However, the IORs of the CORBA objects associated with the agent maintained in the Naming Service still points to the old instance in agent system 1. Thus, the agent updates the references, using the rebind method on the Naming Service.
15. *withdraw:* The agent withdraws its offer in the trader via the trader's Register interface.
16. *export*: The agent export a new offer in the trader via the trader's Register interface. This new offer comprises the new IORs.
17. *remove*: The factory that has successfully created the new agent instance, removes the old instance via the agents LCO interface.

Note: In the scenario mentioned above, just the agent's externalized data is transferred to the new location.

Another important aspect is the transfer of the agent's code, for the case that it is not already available before the migration. At first sight, three possibilities seems to be feasible for the code transfer:

1. The code is transmitted as a parameter of the *create* method invoked on the Factory. The *create* method allows any parameters to be used. For this reason, the code must be packed in a form that can be transferred.
2. The Factory "internally" performs the code transfer, (i.e., in an implementation-specific way). For instance, the code of an agent implemented in Java could be transmitted by using the Java class loading mechanism [38].
3. For this purpose, the CORBA environment must have adequate marshaling routines as well as appropriated factories to handles this date type. This native stream may be

specified as propose within the Objects By Value Service. But this needs more investigation.

4. A new CORBA Service has to be introduced that provides a code loading service.

5.6 Limitations of the current CORBA model

As shown in Section 5.5, the most fundamental operations, (i.e., agent creation, termination, and migration), can be realized by means of existing CORBA Common Services. However, at least the migration scenario provides several disadvantages that could be eliminated by means of enhancements and modifications of the CORBA services. Among others, the following disadvantages can be identified:

- The agent itself has to manage the whole migration procedure, comprising the creation of a stream for containing its externalized data, the localization of a suitable agent system, the creation of its new instance at the destination location, and the modification of its entries in the Naming and the Trader Service. Since a mobile agent's code and data should be as small as possible in order to keep the migration duration acceptable small, as much functionality as possible should be placed outside the mobile agent, (i.e., in the surrounding, static environment).
- An agent that wants to migrate to a specific location requires a suitable agent system at the desired destination side. If no suitable agent system is running, the agent is not able to migrate. Thus, an additional CORBA service is desirable which is able to create agent systems "just in time" when an agent arrives. This service should support the creation of different agent systems (of different manufacturers).
- In the scenario described in Section 5.4.3, many interactions are necessary between the source and the destination location. In a usual CORBA context, this is no problem. One important feature of CORBA is that this architecture allows clients and servers to

interact remotely in a location-transparent way. However, in contrast to this, one fundamental concept of mobile agent technology is to move agents "as close as possible" to their intended communication peers. In this way, MAT reduces the network load and the dependency of network availability. Following this approach, interactions between the source and the destination location should be minimized, even during agent migration.

5.7 Conclusions

The CORBA standard has gained high acceptance in the world of distributed computing. Various service specifications have been developed, providing standardized, implementation-independent interfaces and a common protocol, and thus enabling a high degree of interoperability between applications of different manufacturers. In contrast to this, mobile agent technology is driven by a variety of different approaches regarding implementation languages, protocols, platform architectures and functionality. Thus in this chapter the author has presented various issues and a broad framework required for mobile agent technology in order to achieve a sufficient integration with CORBA. This standard has to handle interoperability between different agent platforms, and the usability of (already existing) CORBA services by agent-based components. In the next Chapter we will study various aspects of such a standard – MASIF.

CHAPTER 6
MOBILE AGENTS AND AGENT COMMUNICATION LANGUAGES

6.1 Introduction

Interoperability is a central issue for both the mobile agents community and the wider agents community. Unfortunately, the interoperability concerns are different between the two communities. As a result, inter-agent communication is an issue that has been addressed in a limited manner by the mobile agents community. Agent communication languages (ACLs) have been developed as tools with the capacity to integrate disparate sources of information and support interoperability but have achieved limited use by mobile agents [54]. In this chapter the author differentiates various issues with regards to inter-agent communication as far as mobile agents and generic agents go, moreover investigates the origins of the differences of perspective on agent-to-agent communication and explores achieving interoperability by use of ACLs in mobile agents paradigm.

There are still two basic questions that are not clearly answered with regards to the mobile agent paradigm: (a) what is an agent, and (b) why do we need mobile agents.

After study, the author comes to the conclusion that the answer to these questions is a matter of perspective, at least until useful applications of mobile-agents come about.

The author will forego giving his perspective on these questions and will instead focus on another question, which has not attracted much attention. Even though agent-to-agent communication is a central issue in the pursuit of interoperability in the agent community, why is it rarely the case that agent communication languages (ACLs) and mobile agents are mentioned in the same context? The author supports the view of software agents [66] as an emerging software-building paradigm. The paradigm definitely introduces a

powerful and ubiquitous abstraction that exhibits the following key concepts: (a) a software agent is an autonomous goal-directed process capable of performing actions [59], (b) is situated in, is aware of and re-acts to its environment, and (c) cooperates with other agents (software or human) to accomplish its tasks.

In the context of the above question, the author adopts the viewpoint that suggest that mobile agents may be thought as programmed entities that can freely roam the network and act on behalf of their users [65]. A slightly more technical definition suggests that mobile agents are programs, typically written in a script language, which may be dispatched from a client computer and transported to a remote server computer for execution [68]. From a historical perspective, the work on distributed computing led to mobile agents following a route through the problem of process migration and the concept of mobile objects [65]. A large part of the work on agents, on the other hand, has its roots in various branches of Artificial Intelligence.

In Section 2, the author examines what interoperability has come to mean for the mobile agent community and the (largely non-mobile) agent community, respectively. In Sections 3 and 4 the author argues in favor of ACL support for mobile agents; the argument is based on the power of an ACL as an interoperability mechanism and tool, and the observation that such interoperability is at least as important for mobile agents as it is for any other kind of agents. After a brief coverage of today's ACLs and their status in Section 5, we discuss reasons why the mobile agent community might be reluctant to fully engage ACLs and explore their integration in mobile agent frameworks and systems.

6.2 Agents and interoperability

Agents are meant to work with other agents. A central point of the agent paradigm of software development is that communities of agents are much more powerful than any

individual agent, which immediately raises the necessity for interoperable agent systems. But the mobile agent community and the agent community at-large do not necessarily refer to the same things; the issues involved regarding interoperability amongst agents are very different. Before exploring the historical roots of these conceptual differences, the author will try to identify the issues that each community associates with interoperability. Briefly, for the mobile agents community interoperability work focuses on:

- The execution environment
- The standardization of some of its aspects and features.

In the case of (non-mobile) agents, there is no notion of an execution environment and the focus is on communication as the means for achieving interoperability.

Mobile agents reside in a highly heterogeneous environment [53]. Mobile agents migrate to a host where an execution environment is set up for them; upon arriving there, they might execute code, make remote procedure calls (RPCs) in order to access the resources of the host, collect data and eventually might initiate another process of migration to another host. While residing on a particular host, mobile agents might have limited interaction (communication) with other agents on the host through an RPC-type mechanism. A potential problem arises from the fact that not all platforms for mobile agents are the same.

6.2.1 The MASIF standard

The Mobile Agent Facility (MAF) proposal [69], which was subsequently replaced by MASIF [63,64], which was dealt with in great detail in chapter 4, is an attempt to standardize some aspects of this execution environment.

MASIF is a collection of definitions and interfaces that provides an interoperable interface for mobile agent systems. MASIF's interoperability is not about language inter-operability but instead it aims at interoperability between agent systems written in the same

language although possibly by different vendors/architectures. MASIF focuses on standardizing three things: (1) agent management, (i.e., standard operations such as creating an agent, suspending it, resuming, and terminate it); (2) agent transfer, (i.e., a common infrastructure for agent applications to freely move among agent systems of different types); and finally (3) names for agents and agent systems. It is expected that the use of a standardized syntax and semantics of agent and agent system names will allow agent systems and agents to identify each other, as well as clients to identify agents and agent systems.

There are issues that are not addressed by MASIF, either because they are outside the scope or because the field is not mature enough for standardization over some issues. One of them is agent communication as it is extensively addressed by CORBA discussed in great detail in chapter 5. As such a viewpoint suggests, agent communication for the mobile agents community means, at least most of the time, the exchange of objects or object references.

6.2.3 3—Layered problem of interoperability

Agent research ought to take into account the following realities: (a) various languages, representing different programming paradigms (procedural, object-oriented, logic, functional, etc.) will be used for implementing agents, (b) hardware platforms and operating systems are expected to be equally varied, and (c) agents are going to be written as autonomous applications and thus few assumptions might be made about their internal structure. A generic break up of the large problem would be into three layers.

One layer is that of translation between languages in the same family (or between families) of languages. This is a very formidable task. The Object Management Group (OMG) standardization effort is an example of work in this direction, within the family of object-oriented languages.

Another layer is concerned with guaranteeing that the semantic content of tokens is preserved among applications; in other words, the same concept, object, or entity has a uniform meaning across applications even if different names are used to refer to it [54].

A third layer relates to the communication between agents. This is not about transporting bits and bytes between agents; agents should be able to communicate complex information and knowledge content.

Agents need to ask other agents, to inform them, to request their services for a task, to find other agents who can assist them, to monitor values and objects, and so on. Such functionality, in an open environment, cannot be provided by a simple remote procedure call mechanism [54].

6.3 A wider interoperability perspective

So, the first question to ask is what happens when a mobile agent needs to interact with mobile agents that are currently hosted in a platform of a different design; in such a case a mobile agent cannot even visit such a platform. Hopefully, a wide acceptance of the MAF/MASIF proposal will help with this problem. But, let us consider the case of a mobile agent that needs to interact with some agent that is not endowed with mobility. In this case, there is no agent platform to visit. Should the mobile agent be prevented from interacting with the static agent or with an information source that is not a full fledged agent but might offer some way of interacting with its information content. Thus it is very clear that interaction between these agents is a must.

Interaction means more than simply exchanging messages [23,24]; it also involves facilities for finding these information sources. Finding, means not only physically locating them on the network, but also been able to identify them or judge them relevant based on their apparent content or capabilities.

There is, finally, another issue of a different nature, which is particular to mobile agents. Mobile agents carry their own code that prescribes the *"how to"* of the tasks they can tackle. This procedural approach towards problem solving can be very limiting [55]. In the current mobile agents paradigm there is not much room for a declarative approach, (i.e., one in which agents simply specify the task they want performed), leaving the details of carrying out the task to the recipient of the request, mobile agents ought to be able to communicate tasks in a common language [54].

6.4 ACLs as an interoperability mechanism

One common abstraction of an agent is as a *Virtual Knowledge Base* [62] – a collection of (mostly) declarative information and knowledge and an associated inference mechanism that allows it to proactively make inferences, answer queries, and appropriately take action. In this context interoperability is often interpreted as a problem of enabling agents with sharing their information and knowledge content.

Three basic problems need to be addressed for agents to effectively share knowledge. First, how we can translate from one knowledge representation language to another; second, how we can guarantee that the meaning of concepts, objects, relationships is the same across different agents; and third, how this potentially sharable knowledge is going to actually be shared, communicated between agents.

An Agent Communication Language (ACL) is a tool that follows the path of this layered abstraction of the interoperability question. An ACL can be thought of as a collection of message types each with a reserved meaning [54]. A communication language is not concerned with the physical exchange, over the network, of an expression in some language, but rather with stating an attitude about the content of this exchange [54].

6.4.1 ACL and a generic messaging protocol

From a software engineering point of view, an agent communication language can be viewed as another messaging protocol, but with two major differences: (a) it describes the application and the actions it can perform, or it can be requested to perform, at a higher level of abstraction, and (b) offers a larger variety of message types. An ACL thus introduces a powerful abstraction because it separates (1) the expressions that are the content of the exchange and (2) their meaning, from the attitude that is expressed about them. ACLs offer a conceptual framework that can assist in addressing the difficult problem of achieving interoperability between applications [54]. Although interoperability is not a problem unique to agents, the very fact that software agents are meant to be, almost by definition, autonomous applications designed with minimal a priori expectations for the state of the rest of the universe brings interoperability to the forefront. Mobile agents are even more susceptible to differences in architecture and language [23,24].

6.4.2 Advantages of an ACL

An agent communication language offers three kinds of advantages. First, an ACL supports interoperability between static and mobile agents, between mobile agents designed for different agents platforms, and also between mobile agents and static agentified information sources. Second, the declarative nature of most ACLs provides many features that make interoperability easier, such as abstracting away some of the lower level, more procedural aspects of the systems involved. Finally, the higher level of abstraction at which ACLs operate can accommodate multiple paradigms.

Thus far, most of the work on mobile agents has focused on the problem of designing the agent platform and addressing major issues surrounding its design, such as security and authentication. Agent communication has been put in the background, partly

because of the concentration on fundamental issues and partly because the ACL alludes to a declarative approach that contradicts the procedural approach implicit in the mobile agent paradigm [55]. There are some exceptions to this situation, of course, such as some experimental work in integrating Agent Tcl and KQML [67].

6.5 Current ACLs

The Knowledge Sharing Effort [70] (KSE) was initiated circa 1990 by DARPA. Its goal was to develop techniques, methodologies and software tools for *knowledge sharing and knowledge reus*e, at *desig*n, *implementatio*n, or *execution* time.

The central concept of the KSE was that knowledge sharing requires communication, which in turn, requires a common language; the KSE focused on defining that common language. Some other agent that uses a different implementation language and domain assumptions should understand expressions in a given agent's native language. The ACL is only concerned with capturing *prepositional attitudes*[3] [56], regardless of how prepositions are expressed. But still, prepositions are what agents will be "talking" about. KIF [61], a particular logic language, was proposed within the KSE as a standard to use to describe things within computer systems, (e.g., expert systems, databases, intelligent agents, etc.). In the next section the author discusses the basic concepts of KQML [59] illustrating the basic concepts of existing ACLs.

6.5.1 Basic concepts of KQML

All KQML dialects and FIPA ACL follow the same basic concepts of KQML that we discuss here. KQML is a high level, message-oriented communication language and

[3]The proper term is *prepositional attitude*s. Prepositional attitudes are three part relationships between: (1) an agent, (2) a content bearing proposition *(e.g.*, "it is raining"), and (3) a finite set of prepositional attitudes an agent might have with respect to the preposition.

protocol for information exchange independent of content syntax and applicable ontology [59]. So, KQML is independent of the transport mechanism (TCP/IP, SMTP, IIOP, etc.), independent of the content language (KIF, SQL, STEP, Prolog, etc.) and independent of the ontology assumed by the content [59].

The KQML language is divided into three layers: the content layer, the message layer, and the communication layer. The content layer bears the actual content of the message, in the programs own representation language. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. Every KQML implementation ignores the content portion of the message, except to determine where it ends. The communication level encodes a set of features to the message, which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication. It is the message layer that is used to encode a message that one application would like to transmit to another. The message layer forms the core of the KQML language, and determines the kinds of interactions one can have with a KQML speaking agent. The primary function of the message layer is to identify the network protocol to be used to deliver the message and to supply a speech act or performative which the sender attaches to the content (such as that it is an assertion, a query, a command, or any of a set of known *performative*s). In addition, since the content is opaque to KQML, this layer also includes optional features, which describe the content language, the ontology it assumes, and some type of description of the content, such as a descriptor naming a topic within the ontology. These features make it possible for KQML implementations to analyze, route and properly deliver messages even though their content is inaccessible.

Though there is a predefined set of reserved performatives, it is neither a minimal required set nor a closed one. A KQML agent may choose to handle only a few (perhaps one or two) performatives. The set is extensible; a community of agents may choose to use additional performatives if they agree on their interpretation and the protocol associated with each. However, an implementation that chooses to implement one of the reserved performatives must implement it in the standard way.

One of the design criteria for KQML was to produce a language that could support a wide variety of interesting agent architectures [59]. Thus, KQML introduces a small number of KQML performatives, which are used by agents to describe the metadata specifying the information requirements and capabilities; KQML also introduces a special class of agents called *communication facilitators* [62]. A facilitator is an agent that performs various useful communication services, (e.g. maintaining a registry of service names, forwarding messages to named services, routing messages based on content), providing "matchmaking" between information providers and clients, and providing mediation and translation services.

6.5.2 FIPA and its ACL

Although the Knowledge Sharing Effort (KSE) introduced the major research issues and approaches to the interoperability problems of the agent's community it did not offer a disciplined way for developing standards. This void was filled in 1996 by the Foundation for Intelligent Physical Agents (FIPA). FIPA is a nonprofit association whose purpose is to "promote the success of emerging agent-based applications, services and equipment." FIPA's goal is to make available specifications that maximize interoperability across agent-based systems. The *modus operandi* of FIPA is to assign tasks to Technical Committees (TCs) that have the primary responsibility for producing, maintaining and updating the specification(s) that are applicable to their task(s). A TC was assigned with the task of

producing a specification for an Agent Communication Language. Along with the TC in charge of Agent Management (agent services, such as facilitation, registration and agent platforms) and Agent/Software Interaction (integration of agents with legacy software applications) they form the backbone of the FIPA specifications.

The FIPA Agent Communication Language (FIPA ACL), like KQML, is based on speech act theory: messages are actions, or communicative acts, as they are intended to perform some action by virtue of being sent. The specification consists of a set of message types and the description of their pragmatics, that is the effects on the mental attitudes of the sender and receiver agents. Every communicative act is described with both a narrative form and a formal semantics based on modal logic. The specification also provides the normative description of a set of high level interaction protocols, including requesting an action, contract net and several kinds of auctions. The FIPA ACL is superficially similar to KQML. Its syntax is identical to that of KQML's except for the different names for some reserved primitives. Thus, it maintains the KQML approach of separating the outer “language” that defines the intended meaning of the message and the inner language, or “content language” that denotes the expression towards which the beliefs, desires and intentions of the interlocutors, as described by the meaning of the communication primitive, apply. The FIPA ACL specification document claims that FIPA ACL (like KQML) does not make any commitment to a particular content language [57].

6.6 Bringing the two worlds together

In this section the author investigates the possibilities of supplementing mobile agents with the ability to handle an ACL. As previously argued, doing so will make a new range of interoperability options available to mobile agents. The deep division between stationary and mobile agents is unnecessary and integrating ACLs into mobile agents

frameworks might not be as hard of a problem. Moreover, the MASIF proposal and the enduring presence of FIPA, along with the continuous interaction between the communities involved in these two endeavors, make the time right for the relatively minor adjustments needed to bring the two worlds together.

6.6.1 Origins of the different perspectives

The semantic approaches of current ACLs [71] suggest the origins of much of the past and ongoing research on ACLs. Rooted in various branches of Artificial Intelligence, these semantic approaches rely on multi-modal logics that are often non-computable and/or have no efficient implementation. Mobile agents, however, are usually programmed in object-oriented or scripting languages. Traditionally, this is not the domain of AI programming languages.

But things seem to be changing. An informal survey of the various multi-agent systems that use an ACL for inter-agent communication would reveal two interesting trends.

- Java is rapidly becoming the language of choice for building agents and knowledge based systems in general [60].
- Many of the new APIs for agent communication languages [72] offer support for modeling, manipulating and reasoning about conversations among agents.

Conversations offer an intuitive way to structure an agent's activities. Also, given the problematic nature of compliance with the ACL's semantic account, conversations shift the focus from the internals of the agent to its observable behavior expressed as sequences of messages sent to other agents. Agents can agree on a conversation protocol for a particular task (e.g., negotiation or auction) and then engage in a scripted interaction. The reluctance of the mobile agents community to engage into the world of ACLs has to do, in some degree, with the programming languages that dominate each branch of the “agent” paradigm. The

fact that Java is becoming the favorite language for programming agents of all types, however, might be changing that. The reality of a language like Java can be seen at the realm of the semantic definition of ACLs.

6.6.2 Integrating ACL's and mobile agents

From a software design point of view, for an agent system to "speak" KQML (or FIPA ACL for that matter) the following things have to be provided:

1. A suite of APIs that facilitate the composition, sending and receiving of ACL messages.
2. An infrastructure of services that assist agents with naming, registration and basic facilitation services (finding other agents that can do things for your agent).
3. The code that for every reserved message type (performative or communicative act) takes the actions prescribed by the semantics, for the particular application; this code depends on the application language, the domain and the details of the agent system that uses the ACL.

Unfortunately, such services were not the focus of the standardization efforts until very recently. There is no service where one can register an agent by just sending a registration message. This problem, which is being addressed by the agents' community, has to do primarily with the lack of agreement on the naming scheme for agents. Because mobile agents are typically programmed in object-oriented or scripting language, it might seem that the major obstacle for adopting ACLs in the mobile agents community is the difficulty of translating the semantics account into code in such languages that complies with the ACL's specification. As Java rapidly becomes the language of choice in the agents' world, the author feels that this will no more be a problem, or at least, it is not more of a problem for

mobile agents than it is for agents in general. This view seems to be supported by the emphasis in conversations and the specification of conversation protocols; this shift of focus suggests a programming-friendly way to account for an agent's communicative behavior.

6.6.3 Problems Integrating ACL's and mobile agents

The first problem is that mobile agents ought to be able to compose, send and receive ACL messages. At this point, they are only concerned with the code necessary for these tasks. One answer to this issue is that mobile agents need not actually have such code; the *agent place* [4]can provide this functionality [5]. As we mentioned, there is a variety of Java APIs that support that kind of functionality; the only issue is integrating them with places' implementations. The mobile agent will use its usual mechanisms of interacting with the place in order to retrieve its messages or to submit messages for sending. It is likely that a mobile agent will only retrieve or submit the content of the ACL message. A place can interact with other places, facilitators and ontology services in order to manage delivery to the appropriate recipient, or language and ontology translation issues [5]. But these aspects of interoperability and the necessary infrastructure to achieve it do not differ from the requirements for non-mobile agents. Of course mobile agents are still left with the task of processing the content of an ACL message, as they see fit. This task, as is the case for non-mobile agents, will depend on the particulars of the agent and its domain. The only real difference is that the compositions, sending and receiving of messages is delegated to the agent place.

A second obstacle is that mobile agents do not have a permanent location on the network. This has repercussions for the naming scheme needed to identify and refer to them

[4]A place is a context in which an agent executes. It is associated with a location, which consists of the place name and the address of the agent system where the place resides [35]

in the network. Technically speaking, the naming scheme is not part of an ACL specification. Agent naming though is viewed as one of the most essential agent management issues that the ACL community is concerned with.

The views of the agents community have been gradually shifting from the earlier viewpoint of a permanent fixed agent location and a symbolic name associated with it, to a more flexible view, where agents may or may not be connected to the network all the time, or may suspend their operation and resume it in some new location or even might have multiple names associated with a unique identity (in such a case these name are related to each other and to the common identity). Such a view seems to include, at least conceptually, the concept of a mobile agent that changes locations carrying (all or parts of) state and data. To the extend that such concepts are reflected in current naming mechanisms and conventions its should be possible to find a common ground between the two communities.

It would be fruitless to get into more detail about naming matters since existing proposals are just proposal and bound to change, but this is exactly the opportunity we ought to take advantage of.

If we were to assign priority to the technical issues mentioned here, the most important concern would be to ensure that naming conventions are consistent between the MASIF proposal and whatever gets adopted in the wider agents community (for example FIPA's choices).

6.7. Conclusions

Software agents offer a new paradigm for building very large scale distributed heterogeneous applications that focus on the interactions of autonomous, cooperating processes that can adapt to humans and other agents. As such, communication is key to realizing the potential of this new paradigm, just as the development of human language was

critical to the development human society and culture. This holds for mobile agents as well as for stationary ones. Agents use an Agent Communication Language to communicate information and knowledge. Some [62] even go so far as to suggest that the use of a rich ACL is a defining characteristic of a software agent, (i.e., a software agent is any process that makes appropriate use an ACL to exchange in-formation). Although this sounds circular, it is not if we stipulate that an ACL should be sufficiently rich to encode a wide range of knowledge and a reasonable set of prepositional attitudes toward sentences in the ACL.

The distinguishing characteristic of mobile agents, in terms of being agents, should not be their ability to move around the network but their ability to act (autonomously) on behalf of their users and eventually perform tasks for them. Agents can achieve high-level interoperability by communication at a higher-level of abstraction involving such concepts as beliefs, goals, expectations and intentions. Currently, mobile agents present a very narrow view of agent communication that does not take full advantage of communication as an interoperability mechanism. The author argues that at this critical period of standardization efforts in the broader area of agents, bringing mobile agents and agent communication together is an opportunity not to be missed.

CHAPTER 7
AN INTRODUCTION TO THE MASIF STANDARD

7.1 Introduction

Mobile agent technology has gained momentum in various application areas, such as electronic commerce, workflow management, and telecommunications. The reason for this are several benefits of this rather new technology, such as asynchronous task execution, reduction of network traffic, robustness, distributed task processing, and flexible on-demand service provision [1]. However, also legacy technologies, such as the traditional client/server paradigm, are still considered as suitable solutions for many distributed application scenarios. Currently, efforts are spent to combine these two approaches in order to realize a unified distributed middleware, supporting both client/server-based remote procedure calls and mobile agents. The recent OMG work on a Mobile Agent System Interoperability Facility (MASIF) [2] specification can be regarded as a milestone on the road toward such a unified middleware, which enables technology and location transparent interactions between static and mobile objects/agents. This chapter illustrates the current state of standardization as described by the OMG MASIF submission.

7.2 Common Conceptual Model

Mobile agents (also called transportable agents) are a relatively new technology that is fueling a new industry. Because the technology and the industry are new, mobile agent systems (for example, Crystaliz's MuBot [8], Dartmouth College's AgentTcl [9,10], IBM's Aglets [11,12], the Open Group's MOA [2], and General Magic's Odyssey [13]) differ widely in architecture and implementation.

The differences among mobile agent systems prevent interoperability and rapid proliferation of agent technology, and have probably impeded the growth of the industry.

To promote both interoperability and system diversity, some aspects of mobile agent technology are being standardized. This chapter provides an introduction on interoperability and describes the terminology of the MASIF submission.

Following the definition of a common conceptual model, sets of features dealt in the MASIF specification to promote interoperability and enhance the functionality of most existing mobile agent systems are discussed. Together, the definitions and features provide a common interoperability base for mobile agent technology [2].

7.2.1 Interoperability

An important goal in mobile agent technology is interoperability between various manufacturers' agent systems. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized [2]. When the source and destination agent systems are similar, standardization of these actions can result in interoperability.

MASIF specification does not standardize interfaces at the language level. Mobile Agent System Interoperability Facilities (also called MAF, an acronym for the original proposal, Mobile Agent Facility) is about interoperability between agent systems written in the same language, but potentially by different vendors and systems that are expected to go through many revisions within the lifetime of an agent. Language interoperability for active objects that carry "continuations" around is technically difficult to achieve. Furthermore, it is not needed, since the support for different languages can be replicated at each node [3].

This specification does not define standardization of local agent operations such as agent interpretation, serialization, execution, or de-serialization. However, these actions are implementation specific, and there is currently no compelling reason to limit agent system implementations to a single architecture.

7.2.1.1 Issues addressed by MASIF currently

Among others, the following functionality is covered by the MASIF-compliant interfaces due to the requirements defined in the MASIF RFP:

- Agent management
- Agent transfer
- Agent and agent system names
- Agent system types
- Location syntax

This section discusses the reasons for standardizing these aspects of mobile agent technology now.

7.2.1.1.1 Agent Management

This topic comprises the creation, termination, suspension, and resumption of agents. The *MAFAgentSystem* interface provides several methods for this purpose, (i.e., the methods *create_agent*, *terminate_agent*, *suspend_agent*, and *resume_agent*).

7.2.1.1.2 Agent Transfer

The topic "agent transfer" states that it is advantageous for two agents to communicate at the same location rather than across a network for two reasons: number of network transactions, and data monitoring [2].

Allowing a source agent to travel to an agent system close to the destination agent system achieves the benefit of locality. If the network between the two agents has low bandwidth, filtering data across the net can be expensive and time consuming when compared to local data transfers.

Data monitoring is a task that can continue for long periods of time. It may be preferable to send a mobile agent to the platform providing the data, rather than use a stationary agent to send periodic inquiries for the latest stock price. For this type of application, using a mobile agent is cost-efficient and resource-efficient [2].

7.2.1.1.3 Agent and Agent System Names

In addition to standardizing operations for interoperability between agent systems, MASIF has also standardized the syntax and semantics of various parameters: agent name, agent system name, and location.

When invoking a management operation, the agent being managed must be identified. Therefore, the agent name syntax should be standardized. Standardized agent name syntax also provides two other benefits. It allows an agent system to quickly determine whether it can support an incoming agent, and it allows two agents to identify each other by name [2].

7.2.1.1.4 Agent System Type and Location Syntax

MASIF enforces standardization of the location syntax so that an agent can access agent system type information from the desired destination agent system, and so that the source and destination agent systems can identify each other. If the agent system type can support the source agent, the agent transfer can happen. It is also important to provide a naming authority (an organization that assigns a unique identifier) for each agent system

type. Ensuring uniqueness of agent system type names prevents two companies from duplicating agent system type values [2].

7.2.1.2 Issues to be addressed by MASIF later

The previous section discussed the aspects of mobile agent technology that are addressed by the MASIF now to ensure agent system interoperability. The OMG group identifies other aspects of the technology that should be standardized.

This section describes some of the areas that will be addressed by the standards body in the future. When an agent takes a multi-hop travel, which travels between more than two security domains (regions, see section 2.2.13, “Regions“), the security issues become complex. Most security systems today deal only with security between two domains, which is single-hop travel. The mobile agent community should delay standardizing multi-hop security of mobile agents until security systems can handle the problem. Today’s mobile agent systems use several different languages (for example, Tcl and Java). Therefore, the effort to convert from one agent encoding to another is too complex. When the serialization formats for agent code and execution state are similar, it should be possible to build standard bridges between different agent system types. In the future, features are likely to be added to MAF that improve interoperability, and minimize the need for bridges.

7.2.1.3 MAF Interoperability Summary

Interoperability is an important goal of the MAF Specification [2]. Agent management allows an agent system to control agents of another agent system. MAF addresses this interoperability by defining interfaces for actions such as *suspending, resuming,* and *terminating* agents. This interoperability is relatively straightforward for most agent systems to implement. Agent tracking permits the tracing of agents registered with

MAFFinders (i.e., Naming Services) of different agent systems. This interoperability is also relatively straightforward for most agent systems to implement [2]

Agent communication is extensively addressed by CORBA as object communication and is considered out of scope for the MASIF specification. Therefore, agent communication is omitted from the MAF specification [2]. MAF addresses agent transport by defining methods for receiving agents and fetching their classes. Agent transport interoperability is not simple to achieve and requires a certain amount of cooperation between implementers of different agent systems to achieve. Note that this chapter presents an overview of interoperability and does not discuss the details of achieving interoperability.

An important goal in mobile agent technology is interoperability between various manufacturers' agent systems. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized. When the source and destination agent systems are similar, standardization of these actions can result in interoperability. However, when the two agent systems are dramatically different, only minimal interoperability can be achieved.

7.2.2 Basic Concepts

This section defines the major mobile agent concepts, some of the concepts have already been introduced in chapter 1, and thus the author will concentrate on concepts specific to the MASIF reference model.

7.2.2.1 Agent Authority

An agent's authority identifies the person or organization for which the agent acts. An authority must be authenticated.

7.2.2.2 Agent Names

Agents require names that can be identified in management operations, and can be located via a naming service. Their authority, identity and agent system type names agents. An agent's identity is a unique value within the scope of the authority that identifies a particular agent instance. The combination of an agent's authority, identity and agent system type is always a globally unique value. Because an agent's name is globally unique and immutable, the name can be used as a key in operations that refer to a particular agent instance.

7.2.2.3 Agent Location

The location of an agent is the address of a place (refer to section 1.2.12, "Place" for more information). A place resides within an agent system. Therefore, an agent location should contain the name of the agent system where the agent resides and a place name. Note that if the location does not contain a place name, the destination agent system chooses a default place.

7.2.2.4 Agent System

An agent system is a platform that can create, interpret, execute, transfer and terminate agents. Like an agent, an agent system is associated with an authority that identifies the person or organization for which the agent system acts. For example, an agent system with authority Bob implements Bob's security policies in protecting Bob's resources.

An agent system is uniquely identified by its name and address. A host can contain one or more agent systems.

7.2.2.5 Agent System Type

An agent system type describes the profile of an agent. For example, if the agent system type is Aglet, the agent system is implemented by IBM, supports Java as the Agent

Language, uses Itinerary for travel, and uses Java Object Serialization for its serialization. This specification recognizes agent system types that support multiple languages, and languages that support multiple serialization methods. Therefore, a client requesting an agent system function must specify the agent profile (agent system type, language, and serialization method) to uniquely identify the desired functionality.

7.2.2.6 Agent System to Agent System Interconnection

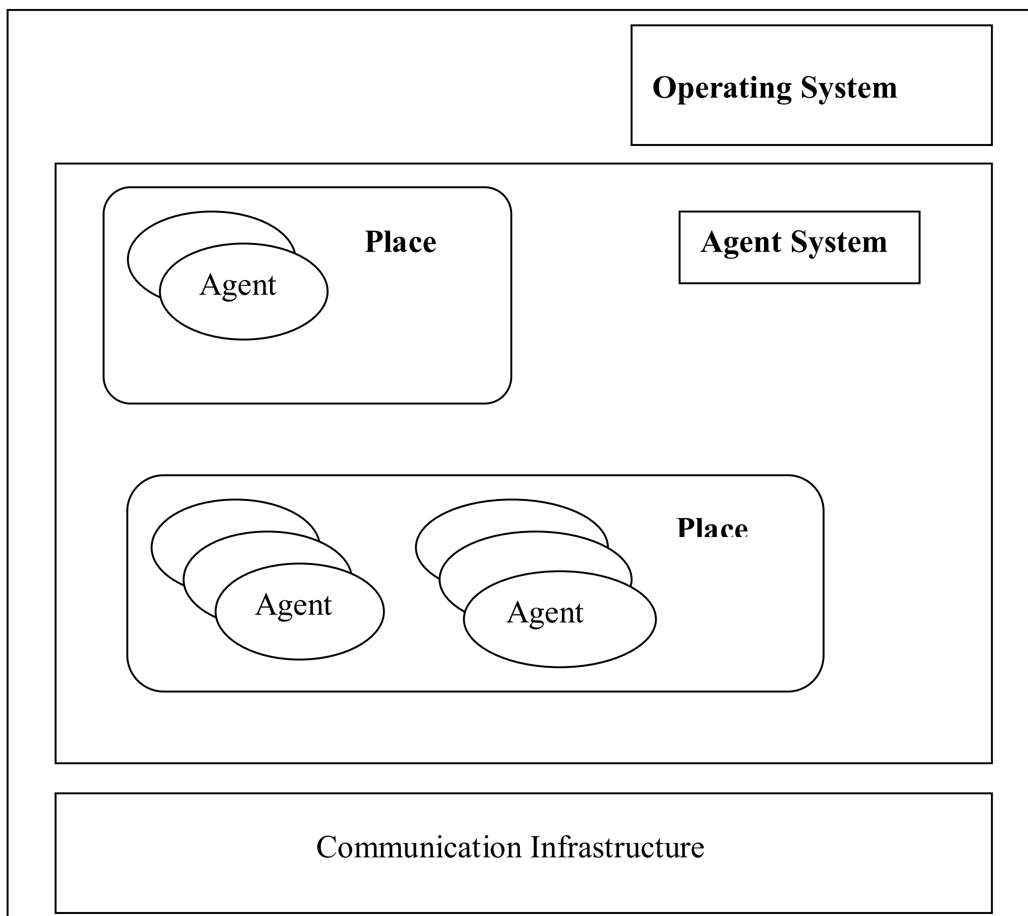

Figure 7.1 Agent System architecture

When an agent transfers it, the agent travels between execution environments called places**.** A place is a context within an agent system in which an agent can execute. This

context can provide functions such as access control. The source place and the destination place can reside on the same agent system, or on different agent systems that support the same agent profile.

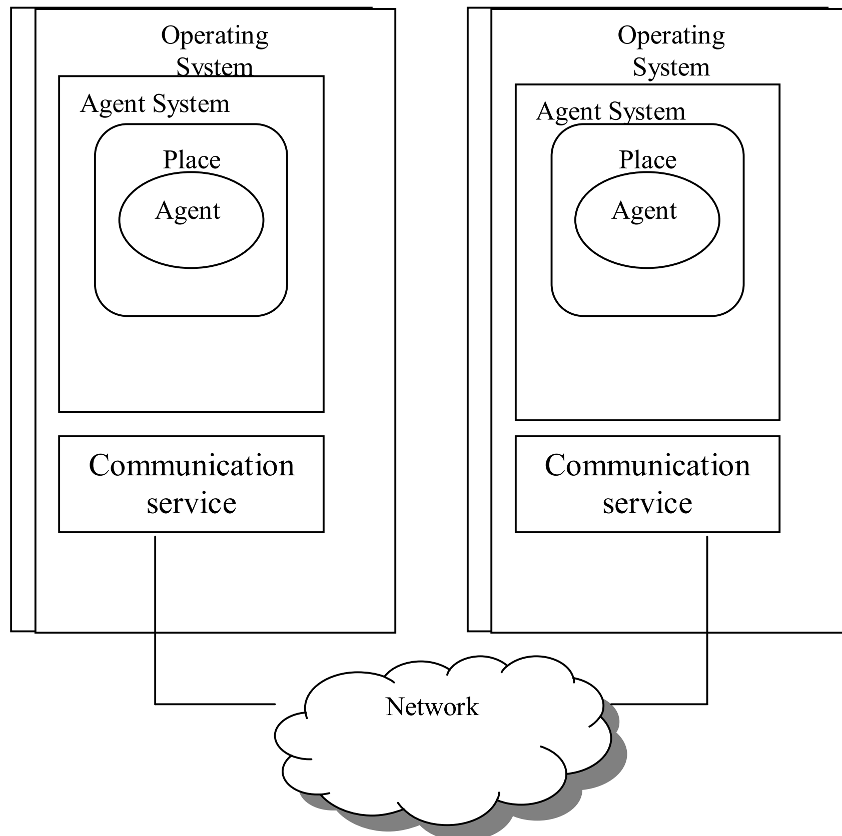

Figure 7.2 Agent System to Agent System Interconnection.

A place is associated with a location, which consists of the place name and the address of the agent system within which the place resides. An agent system can contain one or more places and a place can contain one or more agents. Even though a place is defined as the environment where an agent executes, if agent system does not implement places,

then place is still defined as the default place. When a client requests the location of an agent, it receives the address of the place where the agent is executing.

7.2.2.7 Regions

A region is a set of agent systems that have the same authority, but are not necessarily of the same agent system type. The concept of region allows more than one agent system to represent the same person or organization. Regions allow scalability because you can distribute the load across multiple agent systems.

A region provides a level of abstraction to clients communicating from other regions. A client wishing to contact an agent or agent system may not be aware of the agent's location. Instead, a client has an address for the region (basically, the address of an agent system that is designated as the region access point), and the name of the agent or place. It is now possible to contact and communicate with an agent or agent system with only this information.

An agent may also have the same authority as the region in which it is currently residing and executing. This means that the agent represents the same person or organization as the region. Normally, the configuration of the region may grant a richer set of privileges to such an agent than to another resident agent with a different authority. For example, an agent that has the same authority as the region may be granted administrative privileges. A region can be same as an identity domain of CORBA security if the authority of the region equals to identity of the identity domain.

A region fully interconnects agent systems within its boundaries and enables the point-to-point transfer of information between them. Each region contains one or more region access points and by these means, regions are interconnected to form a network.

7.2.2.8 Region-to-Region Interconnection

Regions are interconnected via one or more networks and may share a Naming Service based on an agreement between region authorities and the specific implementation of these regions. A non-agent system may also communicate with the agent systems within any region as long as the non-agent system has the authorization to do so. A region contains one or more agent systems. Agent systems and clients outside of the region access the region via agent systems that are exposed to the outside world, similar to a firewall situation. These agent systems are defined as region access points. Access rights are allocated to agents, based on the same authority as the region in which they are currently running. Using this definition, a region is regarded as a security domain in the context of MAF.

7.2.2.9 Serialization/Deserialization

Serialization is the process of storing the agent in a serialized form. Deserialization is the process of restoring the agent from its serialized form. The key to storing and retrieving agents is representing the state of an agent in a serialized form that is sufficient to reconstruct the agent. Note that the serialized form must be able to identify and verify the classes from which the fields were saved. For agent systems that are not object oriented, the agent state is the extraction of runtime data for the agent, and classes are the code that implements the agent.

7.2.2.10 Codebase

Codebase specifies the locations of the classes used by an agent. It can be an agent system or a non-CORBA object such as a Web server. If an agent system is responsible for providing the necessary classes, the codebase must have enough information to locate the agent system. Such an agent system is called a *class provider*.

7.2.2.11 Communications Infrastructure

A communications infrastructure provides communications transport services (e.g., RPC), naming, and security services for an agent system.

7.2.2.12 Locality

In the context of mobile agents, locality is defined as being close to the destination agent system either in the same host or the same network.

7.2.3 Agent Interaction

Three common types of agent interactions are defined that are related to interoperability:

- Remote agent creation
- Agent transfer
- Agent method invocation.

Here the author would lay emphasis on agent transfer because this is mobile agent specific

7.2.3.1 Remote Agent Creation

In remote agent creation, a client program interacts with the destination agent system to request that an agent of a particular class be created. A client program is:

- A program in a non-agent system
- An agent in an agent system of a different type than the destination agent system
- An agent in an agent system of the same type as the destination agent system

The client authenticates itself to the destination agent system, establishing the authority and credentials that the new agent will possess. The client supplies initialization arguments and, if necessary, the class needed to instantiate and execute the agent. An agent system can also choose to create agents on its own initiative. The new agent will generally have the same authority as the agent system.

7.2.3.2 Agent Transfer

When an agent transfers to another agent system, the agent system creates a travel request. As part of the travel request, the agent provides naming and addressing information that identifies the destination place. The agent also specifies a quality of communication service required for agent transfer. The quality of communication service is not specified by MAF standard [2], it is left open to agent system implementers to specify it. If the source agent system reaches the destination agent system, the destination agent system must fulfill the travel request, or return a failure indication to the agent. If the source agent system cannot reach the destination agent system, then a failure indication must be returned to the source agent system. When the destination agent system agrees to the transfer, the source agent's state, and authority, security credentials, and, if necessary, its code is transferred to the destination agent system. The destination agent system reactivates the source agent, and then execution is resumed.

7.2.3.3 Agent Method Invocation

An agent invokes a method of another agent or object, if authorized and has a reference to the object. As with agent transfer, the agent specifies the required level for quality of service. Similarly to agent transfer, this is not specified by MAF [2]. The communications infrastructure must invoke the indicated method and return the result of the invocation, or return a failure indication. When an agent invokes a method, the security information supplied to the communications infrastructure executing the method invocation must be the agent's authority.

7.3 Functions of an Agent System

Common actions among agent systems are:

- Transferring an agent, which can include initiating an agent transfer, receiving an agent, and transferring classes
- Creating an agent
- Providing globally unique agent names and locations
- Supporting the concept of a region
- Finding a mobile agent
- Ensuring a secure environment for agent operations

The remainder of this section discusses these areas, defining terms as necessary.

7.3.1 Transferring an Agent

In the previous section, transferring an agent includes initiating the transfer, receiving an agent, and transferring classes. The following sections discuss each of these steps.

7.3.1.1 Initiating an Agent Transfer

When a mobile agent is preparing for a trip, the agent must be able to identify its destination. If the place is not specified, the agent executes in the default place of the destination agent system.

When the location of the destination agent system is established, the mobile agent requests the source agent system for a transfer to the destination agent system. This message is relayed using an internal API between the agent and the source agent system.

When the destination agent system receives the agent's trip request, the following actions are initiated:

- Suspend the agent (halt the agent's execution thread)
- Identify the pieces of the agent's state that will be transferred
- Serialize the instance of the Agent class and state

- Encode the serialized agent for the chosen transport protocol
- Authenticate client
- Transfer the agent

7.3.1.2 Receiving an Agent Transfer

Before an agent is received into a destination agent system, the destination agent system must determine whether it can interpret the agent. If the agent system can interpret the agent, it accepts the agent, then:

- Authenticates client
- Decodes the agent
- Deserializes the Agent class and state
- Instantiates the agent
- Restores the agent state
- Resumes agent execution

7.3.1.3 Class Transfer

Class transfer is the ability to transfer class information from one agent system to another. This ability is a requirement in agent systems that support object-oriented agents. Not all agents are object-oriented programs (for example, AgentTcl). There are three reasons why class transfer is necessary during the life span of a mobile agent.

1. Agent instantiation as a part of remote agent creation (Agent class needed)

When an agent is created remotely by invoking a create operation at the agent system, the Agent class is needed to instantiate the agent. If the Agent class does not exist at the agent system that creates the agent, the class information must be transferred from the source agent system.

2. Agent instantiation as a part of agent transfer (Agent class needed)

After an agent travels to another agent system, the Agent class is needed to instantiate the agent. If the Agent class does not exist at the destination agent system, the class must be transferred from the source agent system.

3. Agent execution after instantiation (classes other than Agent class needed)

After an agent is instantiated due to remote creation or agent transfer, the agent often creates other objects. Obviously, the classes of these objects are needed for their instantiation. If any of these objects' classes are not available at the agent system that creates or receives the agent, they must be transferred from the source agent system. The common conceptual model is flexible enough to support variations of class transfer so that implementers have more than one method available [2]. Specifically, the model supports:

- Automatic transfer of all possible classes

The source agent system (the class provider or the agent sender) sends all classes needed to execute the agent with each remote agent creation or transfer request. This approach eliminates the need for the destination agent system to request more classes. However, automatically sending all classes consumes more bandwidth than necessary if any of the transferred classes are already cached at the destination agent system.

- Automatic transfer of the Agent class only, other classes transferred on demand.

The source agent system sends the class needed to instantiate the agent with each remote agent creation or transfer request. If more classes are needed after instantiating the agent, the destination agent system issues requests for these classes to the class provider.

If the class provider is not directly accessible from the destination agent system, the destination agent system issues the request to the sender agent system by calling *fetch_class* method along with the codebase of the classes. In this case, the sender agent system must be able to locate the requested classes either by using the codebase information, or by sending a further request to another agent system associated with the codebase. The sender agent may have a cache for the classes [2].

This approach does not require the source agent system to determine all possible classes necessary before creating or transferring an agent. It is also more efficient as more classes are cached at the destination agent system. However, the agent creation or transfer request fails if the destination agent system cannot access the source agent system to transfer the necessary classes. This failure could happen, for example, if the source agent system is a portable computer that has been disconnected since the agent creation or transfer request was sent successfully.

- Automatic transfer of the Agent class and on-demand transfer of all other classes when transferring an agent, coupled with transferring all classes automatically when creating an agent remotely.

This approach is a combination of the first two for example a remote agent creation is launched by a client that is not always connected (for example, a laptop computer). Therefore, if all classes are automatically transferred for remote agent creation operations, then the problem of losing access to the classes available at the source agent system may be avoided.

- Transfer a list of the names of all possible classes with the agent creation or transfer request

This approach is another combination of the first two. The source agent system sends a list of class names that includes all the classes necessary to perform the specific agent operation. The destination agent system then requests only the classes on that list that have not been cached. This approach is efficient, but still requires the source agent system to know which classes the agent needs before making the agent creation or transfer request. This technique is most effective when there is cooperation between the agent language compiler and the agent system. For example, the compiler can provide a list of classes associated with each agent so the agent system does not parse the agent's code at run time to get this information.

When an agent system requests a class transfer, the agent system must identify the class to another agent system.

7.3.2 Creating an Agent

For each agent, there is a class from which the agent system instantiates an agent. This class is defined as Agent [2]. To create an agent, an agent system creates an instance of the Agent class within a default place or a place the client application specifies. The Agent class specifies both the interface and the implementation of the agent. To create an agent, an agent system should:

- Start a thread for the agent
- Instantiate the Agent class
- Generate (if necessary) and assign a globally unique agent name that can be authenticated
- Start execution of the agent within its thread.

7.3.3 Providing Globally Unique Names

An agent system must generate a unique name for itself, and the places it creates. It also has to generate a unique name for the agents it creates if required.

7.3.4 Supporting the Concept of a Region

An agent system supports a region by cooperating with other agent systems of the same authority and by supporting a region access point. The region access point is in charge of routing external travel requests to internal agent systems.

7.3.5 Finding a Mobile Agent

When an agent wants to communicate with another agent, it must be able to find the destination agent system to establish communication. The ability to locate a particular mobile agent is also important for agent management. Because a mobile agent travels, an agent name must be unique across all agent systems within regions. Agent systems may provide a naming service based on agent names.

7.3.6 Ensuring a Secure Environment

Because a mobile agent is a computer program that can travel among agent systems, a mobile agent is often compared to a virus. So, it is imperative for agent systems to identify and screen incoming agents. An agent system must protect resources including its operating system, file system, disks, CPU, memory, other agents, and access to local programs.

To ensure the safety of system resources, an agent system must identify and verify the authority that sent the agent. The agent system must also know what access the authority is allowed. The ability to identify the authority of an agent enables access control and agent authentication within an agent system. Another aspect of security is confidentiality. For example, one agent traveling to meet another agent might want to keep both the occurrence of the meeting and the substance of the interaction confidential.

7.3.7 Countering Threats

This section presents several examples illustrated in the MASIF specification how to use security policies to counter threats to the integrity of agent communications. The two general problems covered in these scenarios are denial of service and unauthorized access of data and services. Suppose an agent executes code that attempts to consume all system resources. When resources are not available, services are denied to other agents and possibly to code running on the agent system host. To resist an attack like this, the region administrator can impose resource constraints on agents or code originating from untrusted sources. In most cases, including CORBA, the level of trust assigned to an agent is partially a function of the agent's authority and whether that authority was authenticated.

The trust level may also depend on digital signatures (generated strings or numbers that identify the author), or other techniques. If a security method such as digital signatures is used, an agent can be trusted even if it arrives from an untrusted node. In such cases, the signed pieces can be trusted and only the pieces the untrusted node modified are suspect.

The communications infrastructure is responsible for authenticating the agent's authority. The agent system can supply any other safety services used in this scenario. In addition, the agent language can supply safety features that further enhance security.

Another possible attack on system resources can occur when an agent system is flooded with communications traffic. Usually, the lower-level communications equipment deals with this type of attack. An agent system might have no inherent defense against such an attack.

In an agent-based application, there may be an infinite number of techniques for gaining unauthorized access to data or services. The remainder of this section presents two possible threats to data and service integrity, and the defenses against them. Suppose an

attacker can monitor communications traffic that transports agents and decodes their state data. Once the state data is decoded, the attacker has access to any private information that the agent is carrying. To counter this attack, an agent carrying sensitive information may demand confidentiality services as a condition for transport. If the level of protection required is not available, the agent transport should fail.

Suppose an attacker establishes an agent system that claims to operate on behalf of some trusted authority. Faking the identity of a trusted authority allows the attacker to receive agents that may be carrying information meant for only trusted parties. To counter this type of attack, an agent may demand that it only be transferred to agent systems that are authenticated.

7.3.7.1 Security Service Requirements

This section defines and describes the requirements for secure mobile agent communications, which are:

- Client authentication for remote agent creation
- Mutual authentication of agent systems
- Agent system access to authentication results and credentials
- Agent authentication and delegation
- Agent and agent system security policies
- Integrity, confidentiality, replay detection, and authentication

7.3.7.2 Authentication of Clients

Security services must provide for the authentication of non-agent system client applications. This authentication might be done using passwords or smart cards.

Authenticating a client establishes the credentials of agents that the client launches. Client credentials also determine which security policy is used.

7.3.7.3 Mutual Authentication of Agent Systems

Agent systems, operating without human intervention, must be capable of authenticating each other. This authentication is accomplished by proving that the agent system is in possession of some secret information such as a private key. It may be acceptable for a human to enter a password at the time of the agent system's initialization to authorize the agent system to have access to the secret information.

7.3.7.3 Agent System Access

When agent communication takes place, the destination agent system must sample the credentials of both the agent and the source agent system, and verify their authenticity.

An agent authenticator can use this information to authenticate an agent. The result of the authentication process determines which security policy to apply to the communications between the agent and the hosting agent system.

7.3.7.5 Agent Authentication and Delegation

If an agent is migrating to destination agent system, the agent's credentials must be transferred with the agent if the migration succeeds. The credentials may be weakened depending on the results of the authentication. If the communication mechanism is RMI, the client agent's credentials are passed along to the server agent for charging or auditing. When a client agent makes an RPC call, the client agent's credentials are made available to the server agent. If the server agent makes an RPC call on behalf of the client agent, the server agent should be able to pass the client agent's credentials.

An agent uses its thread of execution to take actions (such as making RPC calls) on its own initiative. When an agent takes such an action, the credentials associated with that action must be those of the agent so that the correct security policy is applied.

7.3.7.6 Agent and Agent System Security Policies

An agent should be able to control access to its methods. The agent or its associated agent system must both set and enforce the access controls. If the agent or agent system's access controls are both self-defined and self-enforced, the source agent's credentials must be available to the destination agent system, because this information is needed for access control decisions.

Alternatively, the agent or agent system set the access controls, then require the communications infrastructure to enforce them. For example, an agent could construct an access control list, and then deliver it to the communications infrastructure for enforcement.

7.3.7.7 Integrity and Authentication

For any communication, the requestor must be able to specify its integrity, confidentiality, replay detection, and authentication requirements. The communications infrastructure must honor these requirements, or return a failure indication to the requestor.

7.4 Conclusions

In order to favor the interaction among different mobile agent platforms, the OMG (Object Management Group) has proposed a standard called MASIF (Mobile Agent System Interoperability Facility) [4,9], whose purpose is to overcome the differences of platforms, by specifying a set of functions and interfaces which all agent platforms need to comply with, so to communicate and interact.

MASIF, in its present form, provides for the functionalities required for the first level of interoperability. This consists of moving agents from an agent system to another, once the information concerning the agent has been transferred; the way the destination agent system manages such information depends on the implementation of the agent system, and has not been dealt with in the MASIF specification.

In fact, it does not standardize some operations for the local management of agents, such as those concerning their interpretation, serialization/de-serialization, and execution. Besides, interfaces are defined at the level of the agent system, rather than being defined at that level of the agent.

The quality of communication service is not guaranteed by the MAF standard, *Agent Transfer* is modeled on the concept that it is advantageous for two agents to communicate at the same location rather than across a network, and an agent has to communicate using a standard protocol wherever it is located, the author sees this as a limitation to the standard, in the next chapter the author proposes and analyses, using scenarios, how a dynamic communication protocol can be realized and can improve the quality of communication.

CHAPTER 8
INTER-AGENT COMMUNICATION USING DYNAMIC INVOCATION INTERFACE FOR A GENERIC MOBILE AGENT PLATFORM

8.1 Introduction

The main reasons why the mobile agent technology is not widely used today are firstly the security loopholes that they open up [1] and secondly the overhead of the inter-agent communication framework. The overhead is because communication is realized over a distributed-object model.

Most of the current mobile agent systems run on interpreter based languages like Java [29,32]**,** the introduction of a middle layer between the OS kernel and the agent framework only increases the overheads incurred. Thus, any way by which these overheads can be minimized is the need of the hour.

In this chapter the author proposes and analyzes inter-agent communication using dynamic invocation interface for a generic mobile agent platform.

The author explains the proposed framework in the context of a complex real world scenario. First the scenario is presented and the problem is established. The current solution to the problem is discussed and later the author goes on to formally analyze the proposed framework in this context.

8.2 A dynamic invocation interface

The concept of mobile agent-based computing has received widespread attention in recent years for its potential to support increased asynchrony and disconnected operations in client-server interactions [1][2]. Mobile agents can be used for information searching, filtering and retrieval, and for various other e-commerce applications on the Web, thus

acting as personal assistants for their clients. Most of the inter-agent interaction still happens in the Intranet itself.

A *regional center*[5] can protect users from malicious agents by providing a single agency as an access point in its Intranet, permitting only secure interactions to the outer world. Inside of the intranet, agents may interact by taking advantage of fast but insecure connections.

The problem with the current communication protocol is that the negotiations with the regional center happen in the same way as with others in the Intranet. Thus, the communication framework is not able to take advantage of the Intranet strategy, and overheads of communicating over a distributed object model are being imposed on all the communication lines.

The author uses the methods in the MAF specification [5], to execute the interaction. When the SC (stationary client) creates the agent, it gives the agent an initial list of sites to visit (its itinerary). Either the SC or the agent can modify this list later, based on information it gathers or receives from an agent system.

The transfer of data from regional center to regional center needs to be secure thus a secure but slow protocol can be used when communicating via the Internet, once at a regional center, the agent system un-marshals the agent credentials and invokes the MAFFinder::lookup_agent_system() and gets the location of the agent system next in the itinerary. If this agent system is within the Intranet, security is not much of an issue. Thus the communication framework is able to dynamically decide, based on certain quality of communication parameters that are specified in the agent credentials, as to what protocol it should use to transfer information.

[5]A regional center is a gateway in an Intranet.

The MAF specification makes neither provisions nor guarantees for the quality of communication [5,67,68]. An agent initiates its migration by contacting its current agent system. The current agent system then invokes the MAFAgentSystem::receive_agent() method on the agent system where the agent wants to go.

The communication protocol is transparent to the agent. It is the agent system that decides the protocol to be chooses, however the agent credentials specify the quality of communication service through their credentials.

The following section lists the various interfaces, through which the agent server could communicate by specifying a QoCS[6]

8.2.1 Different protocols for communication

- **CORBA IIOP:** The CORBA 2.0-compliant Internet Inter-ORB Protocol can be used in all environments that support CORBA, independent of a vendor-specific ORB implementation. It uses the standard-compliant mechanism to connect to an object using a CORBA Naming Service.
- **MAF IIOP:** This protocol is a specialization of CORBA IIOP developed for agent system interaction. It is introduced in the MASIF standard and provides the connectivity between agent systems of different vendors.
- **RMI:** Java Remote Method Invocation (RMI) enables Java objects to invoke methods of other Java objects running on another Virtual Machine (VM). Since this protocol is included in every JDK1.1-compliant VM, all Ajanta agencies support this protocol by default without any further installation or configuration effort.
- **RMI with SSL:** Using this protocol, RMI will run over sockets protected with the Secure Socket Layer (SSL) protocol, which provides a secure transport of all data.

[6]Quality of communication service.

- **Plain Sockets:** The fastest method for remote interactions is to use communication via plain sockets to a specific port of the target host. This technique is robust and avoids the overhead of a distributed object model. Plain socket communication is possible in each Internet-enabled environment and thus ensures a more efficient way to realize communication, especially for interpreter based mobile agent frameworks.
- **Plain Sockets with SSL:** Using this protocol, plain socket connections are protected by SSL. The preconditions for the usage are the same as those mentioned for RMI/SSL.

8.2.2 An Illustration to explain the proposal

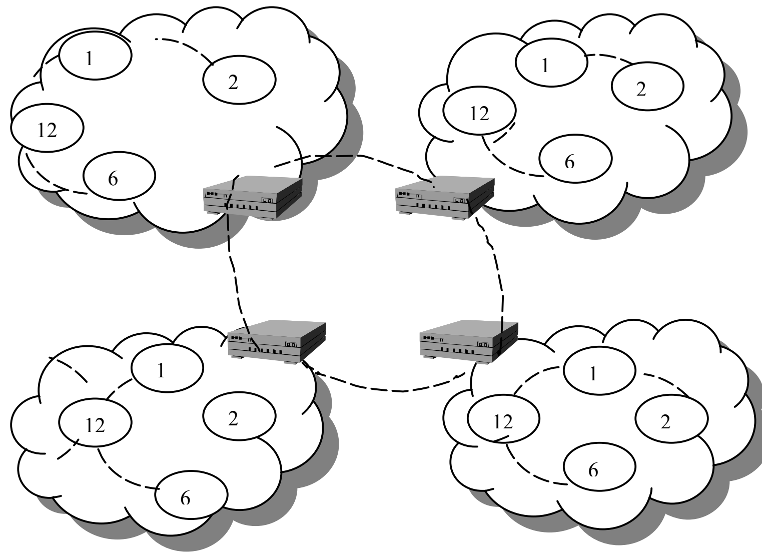

Figure 8.1 Illustration of proposal

If 2 in Intranet 1 wants to get data from 2 and 1 in Intranet 2, previously 2 in 1 would communicate with 2 and 1 in Intranet 2 in the same way. The transfer of data in most of the Java based mobile agent systems is through RMI.

Using the proposed interfaces and infrastructure, all systems in Intranet 2 can communicate by opening up insecure but fast connections like direvt sockets, and thus avoiding the overhead of a distributed object model.

8.3 The scenario

XYZ Inc.[7] is a bookstore, which has grown rapidly and expanded its business into every state, partly by building new stores and partly by acquiring smaller bookstores. They now have a dozen major regional centers, and each one of these has close ties with a dozen or more stores in the region. XYZ offers a wide range of products and services for home and office. Each regional center has its own inventory system, more or less coordinated with the stores in its region. However, part numbers are not fully standardized, and central IT says it will be years before the different systems are fully integrated. Adding to the problem is the continuing fast growth of the store.

8.3.1 Problems being faced

Quality, price, and, above all, service is the XYZ corporate credo. If a customer asks for an item in a store that is not in stock, the CEO insists that the store check with all other stores to see if it is in stock or on order elsewhere. But with their continuing growth and disparate computer systems, this is difficult to do. Sometime, these checks are aggravated by problems, such as a communications link in their network that is down, heavy traffic on their network due to the daily replication and update of the databases in several regional centers,

[7]This is a made up name. It does not refer to any organization.

overheads related to the distributed object model being used to update, query their databases and legacy systems on multiple platforms that have inconsistent part number and vendor number assignments. New stores and new services are not being integrated quickly enough to maintain the level of service the CEO demands, and their customers expect.

8.3.2 The functioning of the store today

A customer comes into the Boston store, and asks for the past 2 years bestsellers in the areas of humor and fiction, for his personal collection. Joe, the salesperson in the store, checks the local inventory on his terminal. He has only 2 of the 10 books in humor and none in fiction. He can check also other regional stores on-line, although that doesn't reflect replenishment orders recently placed, but nothing there either. He calls his contact Susan at another regional center, who looks up the inventory there (no luck), and while Joe is on the phone, she calls two other stores just acquired last month. One of them has the fiction books, but only 5 out of 10. Joe calls three more regional centers without success, though a person at one of the regional centers says he will call back in the afternoon when his computer link is fixed. Joe knows the remaining regional centers are very unlikely to have those books, and he has other customers waiting. He asks the customer to call him tomorrow, and tells him that he should have more information for him then. If they locate the books in another regional center, they can be shipped to this store for pickup in about a week. The customer leaves the store and never calls back. Thus customer satisfaction is the problem the author will be addressing in the following sections.

8.3.3 Solution offered by state-of-art technologies

The IT group of XYZ Inc. has decided that mobile agent technology, combined with better collection and integration of data within a store or regional group of stores, offers

their company the best opportunity for handling rapid growth while maintaining, and, indeed improving, customer service.

Thus, each of the dozen main regional stores in the US is in an Intranet that comprises of approximately a dozen small stores in its proximity. Thus there are 12 Intranets each of which exchanges information via the Internet. Each regional store has installed a mobile agent system; the disparate computer systems forced the IT group to establish different mobile agent platforms for some of the Intranets.

A customer comes into the Boston store, and asks for the past 2 years bestsellers in humor and fiction. Joe, the salesperson in the store, checks the local inventory on his terminal. He finds only 2 of the 20. The terminal tells him it is starting to search other stores to see if they have the other books in stock or on order. Does he want to cancel the search? Does he want information about best sellers in other categories? Joe answers that he does. He is told that he will be given partial results every two minutes, and a chance to modify the search at that time. Two minutes later Joe sees that a store in Atlanta has five of the books, and a store in Charlotte has forty in the horror category. The customer says suspense would be okay, but horror is not. Joe modifies the search criteria and waits. The customer meanwhile has just called his spouse on his cell phone. He is sorry but suspense is not okay. Joe suspends the search. After a few minutes, the customer says the books must be of humor or adventure. Joe modifies the search criteria again, and then tells the system to restart the search from the prior point.

Finally, the requested books were found in Seattle. The customer asks for them to be shipped directly to his home. Joe enters the order, which will be filled by the Seattle store, and shipped UPS.

8.3.3.1 Problems with the current solution

The stores in Dallas, Los Angeles regions and all systems in their Intranets could not be contacted because they were not of a compatible agent system type. The overheads of communicating over a distributed object model were hogging the resources on Joes' as well as systems on all the regional stores as well the smaller stores within them. The IT group was realizing that since all the stores were not yet "mobile", there were stores which could not be accessed because the current mobile agent systems are incapable of using stationary hosts as resources [67,68], moreover they were looking for a way by which quality of communication could also be established, they could no longer be oblivious to these facts and were looking for better solutions to this problem.

8.3.4 High level results

A customer comes into the Boston store, and asks for the past 2 years bestsellers in humor and fiction. Joe, the salesperson in the store, checks the local inventory on his terminal. He finds only 2 of the 20. The terminal tells him it is starting to search other stores to see if they have the other books in stock or on order. Does he want to cancel the search? Does he want information about best sellers in other categories? Joe answers that he does. He is told that he will be given partial results every two minutes, and a chance to modify the search at that time. Two minutes later Joe sees that a store in Cleveland, which could not be accessed before because it used a different mobile agent platform, had all the requested books. The customer asks for them to be shipped directly to his home. Joe enters the order, which will be filled by the Cleveland store, and shipped UPS.

8.3.5 Behind the scenes

Each store has installed a MAF agent system. There are several different hardware platforms involved, but the MAF agent systems interoperate, providing the infrastructure to support mobile agents in a heterogeneous environment.

Within this environment, agents can efficiently find information, and coordinate with each other and the controlling user.

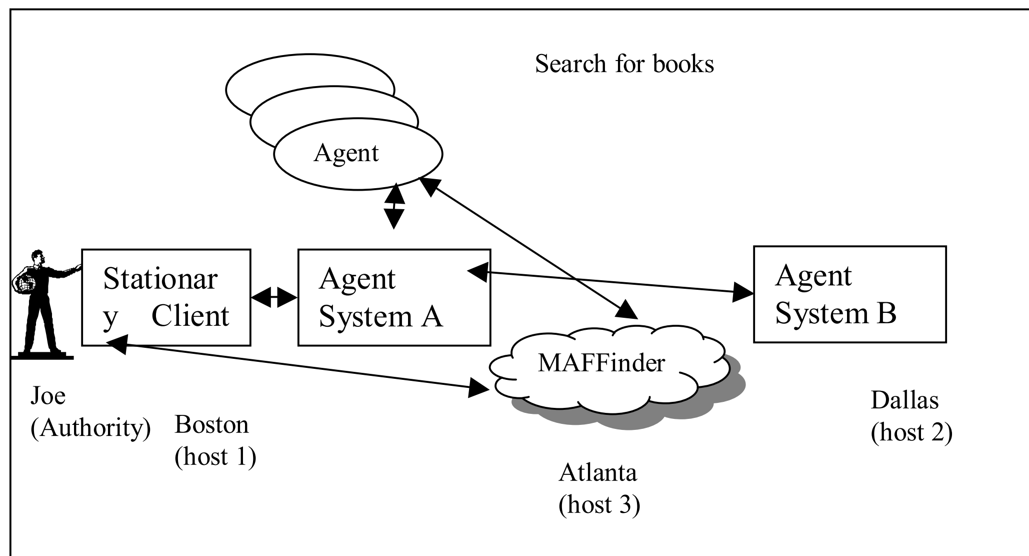

Figure 8.2 Key Actors in the Scenario.

The following entities are involved: ***agents*** *execute on top of* ***agent systems****; agents and agent systems use the* ***MAFFinder*** *to locate other agent systems; the stationary client, using interfaces for an agent system and MAFFinder, can create an agent with Joe's authority to perform specific tasks for it; the stationary client can also monitor and control the agent it created, because XYZ's policy allows agents of the same authority to manage each other. The agent system and MAFFinder interfaces can be invoked by the following*

'clients': a mobile agent, or a stationary client (user application) that monitors/controls agents, or other agent systems.

8.3.5.1 The MAF services this scenario uses

- Locating an agent system for agent creation (using the MAFFinder interface).
- Creating an agent to begin the query (using the MAFAgentSystem interface).
- The agent registers as it moves (using the MAFFinder interface), so that others can find it.
- The agent goes from place to place (using the MAFAgentSystem interface).
- Finding the search agent to request status (using the MAFFinder interface).
- Suspending or resuming the agent (using the MAFAgentSystem interface).

The main actors in the scenario are described in **Figure 5-1**. Joe our salesperson is the authority. The user application interacting with Joe at his terminal is the stationary client (SC). The SC has control over and monitors all agent activity in this scenario.

8.3.5.2 Description of low level details

Joe's request starts a series of actions. The user application (SC) working on Joes' behalf invokes the MAF services to dispatch and control mobile agents going to the appropriate sites.

In all cases where a MAFFinder is used to locate a stationary object such as stationary agents, agent systems, and places, the client actually can choose between the CORBA naming service and the MAFFinder, thus this framework is also able to interact with stationary agents and use them as resources. In the case where a MAFFinder should be located, one can get the object reference of the MAFFinder via MAFAgentSystem::get_MAFFinder(), or via the CORBA naming service.

Joe is informed that the item is not in stock locally. Thus, the SC deploys an agent in the network to search for the same.

8.3.6 Locate

The SC begins by obtaining a reference to a 'finder' (MAFFinder) by invoking a get_MAFFinder() method on behalf of an agent system (MAFAgentSystem). This method takes no parameters. A finder can be shared among multiple agent systems.

8.3.6.1 Details of the above action

The SC obtains the locations of available agent systems and places using the MAFFinder methods lookup_agent_system() and lookup_place(). This information is used to decide where to start an agent, and where to move it during its lifetime. To locate other agent systems and places, the SC needs to know only their names, which are provided as parameters to the methods lookup_agent_system() and lookup_place().

The author proposed to overload these methods to obtain information as to which Intranet the target agent system belongs and stores it in a parameter my_location.

The SC can also obtain additional information about each agent system by invoking the get_agent_system_info() method (MAFAgentSystem interface). This information is agent-system specific. This method takes no parameters, and it returns agent-system-specific information. The lookup operation of an agent system is based either on the agent system name or the agent system information.

When a new agent system starts up, it registers itself with the MAFFinder using the register_agent_system() method. This method takes the agent system name, agent system information, and the location as parameters. The agent system information parameter (agent_system_info) is used during lookup to identify agent systems with specific

characteristics. It is an internal matter of the MAFFinder to announce itself available to other MAFFinders, as well as to announce availability of the new agent system.

Thus apart from using the MAF interfaces which allow locating mobile agent systems of any framework, the location (Intranet) specific information is also obtained.

8.3.7 Create

Based on the availability of other agent systems and places, the SC creates an agent on a selected agent system by invoking the create_agent() method of the MAFAgentSystem interface.

As a part of this invocation, the SC provides various parameters. The agent_profile parameter specifies the agent and the originating agent system specific information, such as the type of the agent system, and the way the agent was serialized. The agent parameter contains agent state, such as the serialized state of an agent's objects, and agent class definitions, the location specific information is also specified here, thus, apart from knowing agent system specific information the agent_profile contains location specific information my_location. The receiving agent system uses this data to create a new instance of the agent. If specified, the place_name is the destination of the agent on the receiving host. Otherwise, the agent is associated with the agent system's default place. The Arguments parameter is passed to the agent constructor. class_names, code_base and class_provider and are used to retrieve agent classes. The create_agent() method returns the name of the agent created.

8.3.8 Register

Once a new agent is created, the SC (as well as the agent) has many interfaces available to control the activity and movement of the agent throughout its life. The agent can also control the activity of other agents.

To allow other SCs and other agents to be aware of it and locate it, the agent needs to register itself with the MAFFinder (if the controlling SC wants registration). This is achieved using the register_agent() method of the MAFFinder interface, which takes as parameters the agent name, agent profile and the agent's current location. Agent profile specifies agent characteristics that can be useful for filtering during a lookup operation. Similarly, if the agent does not want to publicize itself anymore, it can remove itself from the MAFFinder, by invoking unregister_agent(). This method takes only the agent name as a parameter.

The SC acting on behalf of Joe wants the agent to go to each regional center, and get information about relevant stock items.

8.3.9 Move

When the agent completes its work at the first site (host), it migrates to the next host on its list. In this section the author discusses the multi protocol based communication framework that was proposed in (section 2) the advantages of this add-on to the current MAF specification are also illustrated.

When the SC (stationary client) creates the agent, it gives the agent an initial list of sites to visit (its itinerary). Either the SC or the agent can modify this list later, based on information it gathers or receives from an agent system.

At a regional center, the agent system un-marshals the agent credentials and invokes the MAFFinder::lookup_agent_system() and gets the my_location parameter of the agent system next in the itinerary. If this agent system is within the Intranet, security is not much of an issue. Thus a fast and robust mechanism such as communication via sockets can be used.

The communication framework is able to dynamically decide, based on certain quality of communication parameters that are specified in the agent credentials, as to what protocol it should use to transfer information.

An agent initiates its migration by contacting its current agent system. The current agent system then invokes the MAFAgentSystem::receive_agent() method on the agent system where the agent wants to go.

Finally, the agent_sender represents the agent system the agent is leaving (source). When attempting to receive an agent, the destination needs to fetch from the source agent system the classes on an ad-hoc basis. This is achieved using the method fetch_class(), which uses as parameters class_names and code_base. Previously, these parameters were passed to the destination host as parameters of the receive_agent() method.

After a set period of time (about two minutes), the SC wants to retrieve the information the agent has gathered so far, and then pass it on to Joe.

8.3.10 Status

The agent has visited several sites so far, and gathered relevant information from those sites about the particular stock items. Some of these stock searches done locally by the agent are in fact quite complex, processing a lot of data from several databases on interconnected systems, and making a summary of the requested information for later transmission to the SC.

The SC must find and interrogate the agent for its status, and also obtain any relevant data that the agent wants to send back to the SC through the non-MAF interfaces. The SC obtains the status by invoking get_agent_status(), and using as a parameter the name of the agent in question.

The SC can also find out all the places and agents residing at an agent system by invoking the methods list_all_agents() and list_all_places(), respectively. These two methods return the lists of the agent and place names. It is also possible to obtain all the agents that belong to a user (distinguished by its principal) by invoking the method list_all_agents_of_authority(), passing the authority as a parameter.

In some cases, however, it is required that the invoking client be authenticated prior to being allowed to invoke this method. Prior to being able to invoke these methods, the client needs to resolve the agent's current location, by invoking the lookup_agent() method (MAFFinder). The agent must maintain this information by updating appropriate MAFFinders.

Lookup can be based either on agent name or on agent profile. Joe modifies the search criteria and waits.

8.4 Contacting static/un-reachable agent systems

The SC sees from the data that the agent was not able to visit the Miami site. It was on the itinerary, but the agent system in Miami was not able to receive agents at that time. The SC therefore locates another store in the Miami region that has a compatible agent system, and instructs the agent to go there. In order to access an object on an agent system that cannot receive the agent, the agent can invoke a method find_nearby_agent_system_of_profile() on the incompatible agent system. This method takes as a parameter the profile describing the agent system. It returns the location of the closest agent system of compatible type, if this information/system is available. This type of information is useful for improving the locality of reference between the object and the agent in question.

Using the agent name, the SC suspends the agent by invoking the method suspend_agent(). Note that clients other than the SC (the owner), need to be authenticated and authorized to be able to invoke the suspend() and resume() methods.

After four minutes the terminal displays information that a store in Cleveland, which could not be contacted before, has all the required books.

The customer asks for them to be shipped directly to his home. Joe enters the order, which will be filled by the Dallas store and shipped UPS the next day.

The mobile agent has fulfilled its duty, so the SC (or other authorized client) can now terminate it, by invoking the terminate_agent() method. This is the end of the agent life cycle.

CHAPTER 9
A DECENTRALIZED VIDEO CONFERENCE APPLICATION

9.1 Introduction

This chapter is a design overview of a multimedia application for videoconferencing over the Internet. The chapter tries to provide a thorough discussion of the capabilities, specifications and limitations of the mobile agent based Decentralized Video conference (DVC) application. The DVC application is a video conferencing tool to perform real time conferencing on a network using mobile agents. The audio/video media will be streamed between two users over the Internet. The mobile agent system provides the premise for the users to find each other on the network. This system brings a whole new perspective to conferencing systems by decentralizing the DVC server. There is a paradigm shift from execution program on a single server to execution of the program on the various clients or users that use the system.

Video conferencing applications all over the world use the centralized approach [1]. This means that a user who wants to chat has to connect and logon to a central server on which the other user also has to connect and logon. All the execution happens on the server and only the clients or the users give the input and get the output. This places a tremendous load on the server that not only has to provide the necessary support but also has to route all the traffic from the client to itself and from itself to the other client. This causes congestion and has drawbacks that you might well expect in a centralized environment. There is also a single point of failure.

Within the past few years, there has been a rapid growth in network traffic. All kinds of data are transferred on the networks today. This includes media like (sound, video,

images etc.). Each of the different types of data has some bandwidth requirements to perform efficiently on these networks. QoS allows networks to use their existing resources efficiently and to guarantee that critical applications receive high-quality service without having to expand quickly or even provision their networks. QoS parameters can be listed as bandwidth, latency, jitter and reliability. Bandwidth is the rate at which an application's traffic must be carried by the network. Latency is the delay that an application can tolerate in delivering a packet of data. Jitter is the variation in latency. Reliability is the percentage of packets discarded by the router.

Regardless of the delivery mechanism used, all processes of delivering multimedia data on the Web are basically similar; data from the source must be converted to a digital format, which is then encoded. In this process the data is compressed into a more manageable delivery size. When providing content for low-bandwidth encoding, it is crucial to provide as high a quality of source material as possible. It is important to realize that the compression used for encoding is lossy; meaning a great deal of information is removed from the original signal to obtain a particular compression. By providing a high quality source, the encoder will have more meaningful data to analyze that helps produce a noticeably better end product.

The author discusses a solution to the stated problems of centralization and quality of service by providing a multimedia application, which is not centralized. The author uses the mobile agent paradigm to do this.

Section 9.2 gives system architecture of the DVC application.

9.2 System Architecture

The system architecture for the video conferencing application using Mobile Agents is as shown in figure 9.1

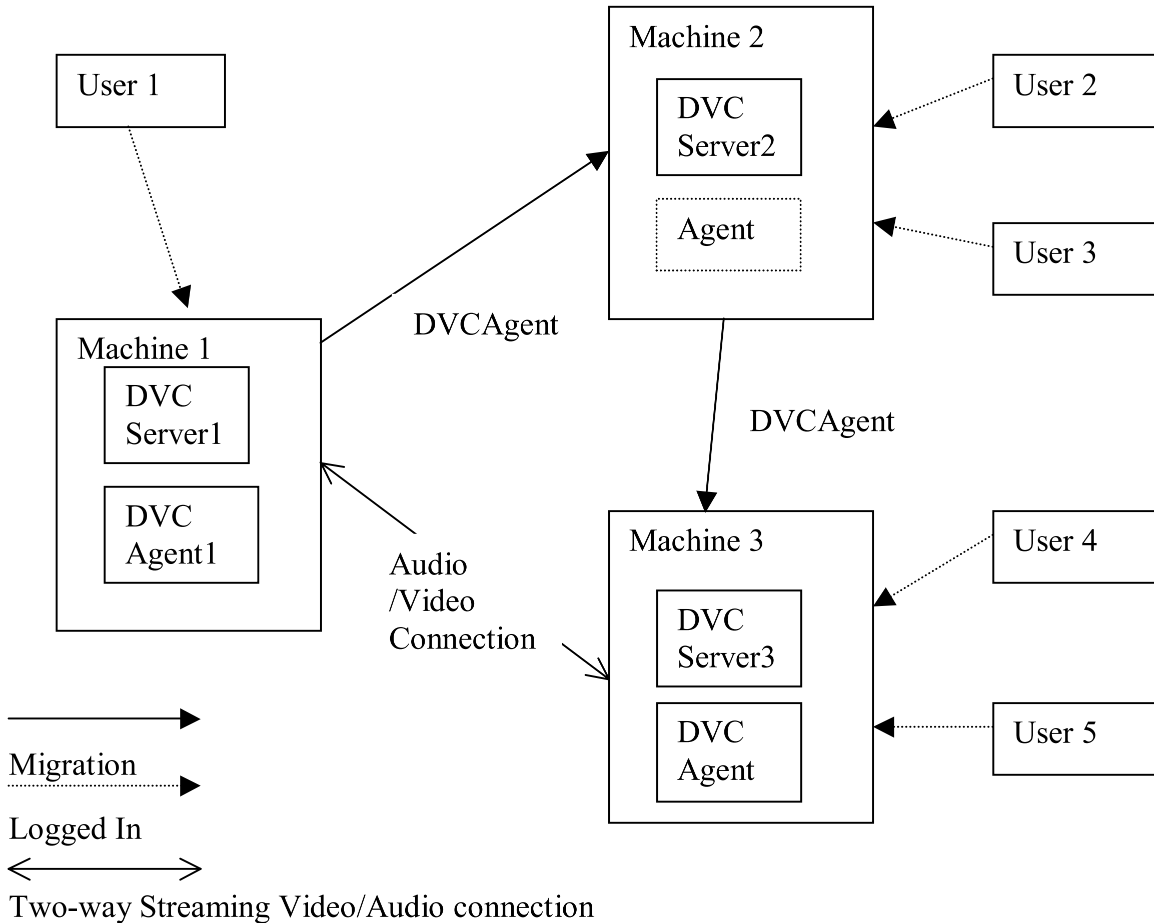

Figure 9.1 Decentralized video conference architecture

The DVCServer extends from the Aglets AgentServer class and the DVCAgent extends from the Aglets Agent class. Users are logged on to various machines running the DVCServer. This means that the DVCServer at any point of time can find out all the users logged onto the machine. The DVCServer keeps track of all users who are logged onto the

machine. It also provides the environment for a DVCAgent to execute its functionality in. As mentioned earlier, the DVCServer is a derivative of the Agent Server class in the Mobile Agent System we are using. DVCAgents can execute only under the jurisdiction of the DVCServer. This gives the DVCServer complete control over the execution of the DVCAgent, thus securing the machine from any malicious agents. The DVCAgent is a piece of code that has a particular itinerary to execute. It travels / migrates from one machine to another and executes the same piece of code there under the vigil of the DVCServer. The DVCAgent has an understanding with the DVCServer and this is provided so that the DVCAgent is protected from malicious Servers. The Agent migrates from machine to machine inside a fixed domain. This domain is the same as specified in the name registry of the mobile agent system.

According to figure 9.1, all the machines in the network are running the DVCServer. User 1 is connected to Machine 1, User 2 and User 3 to Machine 2 and User 4 and User 5 to Machine 3. These users are registered with the DVCServer. A user creates the DVCAgent if he/she wants to talk or chat with another user. The original DVCAgent then clones itself and migrates to other machines across the domain to find the user to talk to. The DVCAgent has the functionality to open a streaming audio and video connection to another DVCAgent. It then gets the voice data from the user through a mike and video data through a camera, compress it and send it to the DVCAgent with the other user which uncompresses the data and feeds it to the user's sound and video cards so that he/she can respond. But first the Agent has to search for the user to talk to.

The User who initiates the audio session tells the DVCAgent that he/she wants to talk to a user by specifying his/her name. The DVCAgent communicates with the DVCServer on the initiating user's machine and finds out whether the user to be talked to is

logged on in this machine or not. If the user is logged on then the DVCAgent opens a streaming audio/video connection between the two users on the same machine. If the user is not logged on, then the DVCAgent clones itself and migrates to another machine in the network domain. There it communicates with the DVCServer running on that machine to find out whether the user to be talked to is logged on in this machine. If the user is logged in then the DVCAgent opens a streaming audio connection with the original DVCAgent in the initiating machine. If the user is not logged in then the DVCAgent migrates to yet another machine where it does the same things as before till it has covered all machines or found the user to talk whichever is earlier.

For example let us say that User 1 on Machine 1 wants to talk to User 4. So User 1 starts up a DVCAgent 1 to do the audio chat. This DVCAgent 1 communicates with DVC Server1 and finds whether User 4 is logged on in Machine 1. After finding out the User 4 is not logged on, DVCAgent 1 clones itself to a DVCAgent. This DVCAgent migrates from Machine 1 to Machine 2. In Machine 2, it communicates with DVC Server2 and finds out whether User 4 is logged on Machine 2. After not finding User 4 logged in, the same DVCAgent migrates to Machine 3. In Machine 3, it communicates with DVC Server3 and finds that User 4 has indeed logged on in Machine 3. Now the DVCAgent opens up a streaming connection with DVCAgent 1 on Machine 1. Once the connection is setup, the DVCAgent does the (sampling, compression, transmission, decompression, playing etc.) as required by the modalities of the chat.

9.3 Conclusions

As networks continue to expand and interact, the problems involved in controlling them from an enterprise point of view expand accordingly, thus solutions such as the DVC become essential to decentralize the network. In addition, the need to gather effective

information on a timely basis is a critical factor. Two trends are obvious: the need to simplify the users' contact with the network and the need to simplify the technical management of the network, including bandwidth management, and the expansion and introduction of new technologies. The future of the Internet depends on its ability to transmit multimedia data over it; the mobile agent paradigm, by its inherent nature saves a lot of bandwidth, and can achieve the above requirement, and the above application shows the new frontiers that the mobile agent paradigm opens.

CHAPTER 10

CONCLUSIONS

10.1 Conclusions

In this thesis the author discussed various issues with regards to interoperability between disparate agent systems. A framework to achieve inter-agent communications through dynamic invocations interfaces has been proposed. Various solutions to the interoperability problem have been explored.

One of the solutions that have been discussed is integrating Agent communication languages (ACLs), which have been developed as tools with the capacity to integrate disparate sources of information and support interoperability, and the MASIF standard.

The author investigates the possibilities of supplementing mobile agents with the ability to handle an ACL.

As part of another solution to the interoperability problem the author gives an overview as to how Mobile agents (MAs) can make use of already defined CORBA objects and services and vice versa offer their services to other CORBA objects.

The state-of-art in agent mobility and communications has been established. Most of the mobile agent systems are being implemented over interpreter based languages like Java thus increasing overheads of the communication mechanism, by making use of different communication protocols for different localities, the author has discussed many advantages that can be gained.

Finally the author described a multimedia application, which is a decentralized videoconference.

10.2 Future Work

The proposed communication framework could be extended to become a plug-and-play kind of an interface, which should not be restricted to certain number of protocols. The multimedia application of videoconferencing can be enhances to a multi-point conference. The capabilities can further be extended to include fault tolerance. Various modes of communicating or passing data like promiscuous, non-promiscuous etc. can be used to deliver data.

REFERENCES

[1] D.C. Chess, C. Harrison, A. Kershenbaum - Mobile Agents: Are They a Good Idea?,IBM Research Report, 1995

[2] T. Papaioannou, J. Edwards - Using Mobile Agents To Improve the Alignment Between Manufacturing and its IT Support Systems, 1998

[3] P. Siepel, J.J. Borbink, B.M.A. van Eck - Intelligent Software Agents: Turning a Privacy Threat into a Privacy Protector, April 1999

[4] T.Taka, T.Mizuno, T.Watanabe – A Model of Mobile Agent Services Enhanced for Resource Restrictions and Security, Proceedings of the International Conference on Parallel and Distributed Systems, 1998, p. 274-281

[5] GMD FOKUS - MASIF – Mobile Agent System Interoperability Facilities Specification, ftp://ftp.omg.org/pub/docs/orbos/97-10-05

[6] P. Siepel, J.J. Borbink, B.M.A. van Eck - Intelligent Software Agents: Turning a Privacy Threat into a Privacy Protector, April 1999

[7] T. Taka, T. Mizuno, T. Watanabe – A Model of Mobile Agent Services Enhanced for Resource Restrictions and Security, Proceedings of the International Conference on Parallel and Distributed Systems, 1998, p. 274-281

[8] Sankar Virdhagriswaran http://www.crystaliz.com/logicware/mubot.html - White paper of Crystaliz, Inc., defining mobile agent technology.

[9] Maes, Pattie (1990) ed., Designing Autonomous Agents, Cambridge, MA: MIT Press

[10] Maes, Pattie (1995), “Artificial Life Meets Entertainment: Life like Autonomous Agents,” *Communications of the ACM*, 38, 11, 108-114

[11] Smith, D. C., A. Cypher and J. Spohrer (1994), “KidSim: Programming Agents Without a Programming Language,” *Communications of the ACM*, 37, 7, 55-67

[12] Hayes-Roth, B. (1995). “An Architecture for Adaptive Intelligent Systems,” *Artificial Intelligence:* Special Issue on Agents and Interactivity, 72, 329-365.

[13] IBM's strategy white paper available at http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm.

[14] Wooldridge, Michael and Nicholas R. Jennings (1995), "Agent Theories, Architectures, and Languages: a Survey," in Wooldridge and Jennings Eds, Intelligent Agents, Berlin: Springer-Verlag, 1-22

[15] Brustoloni, Jose C. (1991), "Autonomous Agents: Characterization and Requirements," Carnegie Mellon Technical Report CMU-CS-91-204, Pittsburgh: Carnegie Mellon University.

[16] Stan Franklin and Art Graesser Institute for Intelligent Systems, University of Memphis Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[17] Russell, Stuart. J. and Norvig, Peter (1995) "Artificial Intelligence: A modern approach " Eaglewood cliffs, NJ: Prentice Hall, pp 33.

[18] Chang, IBM (1998) http://activist.gpl.ibm.com:81/WhitePaper/e2.html

[19] Sunsted, Todd "Agents on the move. Java world July 1998." http://www.javaworld.com/javaworld/jw-07-1998-/jw-07-howto.html

[20] Sunsted, Todd "An introduction to agents" Java World, June 1998.

[21] WJ Buchanan, M Naylor and AV Scott Napier University, Edinburgh, UK "Enhancing Network management using mobile agents."

[22] James W. Stamos and David K. Gifford. "Remote Evaluation". ACM Transactions on Programming Languages and Systems, 12(4): 537-565, October 1990.

[23] John Vittal. Active Message Processing: Messages as Messengers. In R.P. Uhlig, editor, Computer Message System, pages 175-195. North Holland, 1981.

[24] J.S. Banino. "Parallelism and Fault Tolerance in Chorus". Journal of Systems and Software, pages 205-211, 1986.

[25] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-Grained Mobility in the Emerald System". ACM Transactions on Computer Systems, 6(1): 109-133, February 1988.

[26] James E. White. Mobile Agents. Technical report, General Magic, October 1995.

[27] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. “Operating System Support for Mobile Agents”. In Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS-V), pages 42-45, May 1995.

[28] Gunter Karjoth, Danny Lange, and Mitsuru Oshima. “A Security Model for Aglets”. IEEE Internet Computing, pages 68-77, July August 1997.

[29] ObjectSpace. ObjectSpace Voyager Core Package Technical Overview. Technical report, ObjectSpace, Inc., July 1997. http://www.objectspace.com/.

[30] Neeran Karnik and Anand Tripathi. “Design Issues in Mobile Agent Programming Systems.” IEEE Concurrency, July-Sep 1998 pp 52-61.

[31] OMG: The Common Object Request Broker: Architecture and Specification. Technical Report, Revision 2.2, Feb 1998.

[32] Crystaliz Inc, General Magic Inc, GMD FOKUS, IBM, TOG: OMG Joint Submission "Mobile Agent System Interoperability Facility", November 1997, available at ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf

[33] OMG: CORBA Services: Common Object Services Specification, November 1997.

[34] OMG: Objects by Value, Joint Revised Submission, OMG TC Document orbos/98-01-01

[35] Thomas Magedanz, Markus Breugst, Sang Choy: “Integrating Mobile Agent Technology and CORBA, Towards CORBA object mobility” Draft 2.

[36] Thomas Magedanz, Markus Breugst, Sang Choy: “A CORBA Environment Supporting Mobile Objects” Draft 1, January 1999.

[37] OMG: CORBA Components, Joint Initial Submission, OMG TC Document orbos/97-11-24, November 1997.

[38] Sheng Liang, Gilad Bracha: ”Dynamic Class Loading in the Java Virtual Machine”, Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98).

[39] F.G. McCabe, K.L. Clark April - Agent Process Interaction Language; F.G.; Intelligent Agents: Theories, Architectures, and Languages (LNAI volume 890)

[40] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, MA, December 1995.

[41] Shankar Sundaram. Mission-Flow Constructor: A Workflow Management System Using Mobile Agents**.** Masters Thesis, Thayer School of Engineering, Dartmouth College, May 2000.

[42] IKV++GmbH Informations und Kommunikations systeme. Grasshopper: The Agent Platform - Technical Overview. IKV++ GmbH Informations- und Kommunikations systeme, February 1999.

[43] Lange, D. B. et al., Aglets: Programming Mobile Agents in Java, In *Proc. of Worldwide Computing and Its Applications (WWCA '97),* Lecture Notes in Computer Science, vol. 1274, Springer Verlag, 1997.

[44] Voyager technical overview. ObjectSpace White Paper, ObjectSpace, 1997.

[45] M. Breugst, I. Busse, S. Covaci, and T. Magedanz. Grasshopper – A Mobile Agent Platform for IN Based Service Environments. In *Proceedings of IEEE IN Workshop 199*8, pages 279–290, Bordeaux, France, May 1998.

[46] Mitsuru Oshima, Guenter Karjoth, and Kouichi Ono Aglets Specification 1.1 Draft, September 8th, 1998.

[47] McCabe F.G. and K.L. Clark (1994) "*Programming in April*", Department of Computing, Imperial College, London.

[48] F.G. McCabe and K.L. Clark. "Programming in April". Technical report, Department of Computing, Imperial College, London, 1994.

[49] Odyssey: Beta Release 1.0, 1997. Available as part of the Odyssey package at http://www.genmagic.com/agents/.

[50] http://java.sun.com

[51] Crystaliz Inc, General Magic Inc, GMD FOKUS, IBM, TOG: OMG Joint Submission "Mobile Agent System Interoperability Facility", November 1997, available via ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf

[52] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E.Wiegand: "*Agent-based business process management*." International Journal of Cooperative Information Systems, 1996.

[53] Robert S. Gray and George Cybenko and David Kotz and Daniela Rus. Mobile agents: Motivations and State of the Art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*, AAAI/MIT Press, 2001.

[54] Yannis Labrou, Tim Finin, and Yun Peng, `*The current landscape of agent communication languages*', IEEE Intelligent systems, 1999.

[55] D. Rus, R. Gray, and D. Kotz. *Transportable Information Agents*. Proceedings of the first ACM international conference on Autonomous agents, 1997.

[56] R. Patil, R. Fikes, P. Patel-Schneider, D. Mckay, T. Finin,T. Gruber and R. Neches: "*The DARPA Knowledge Sharing Effort: Progress Report*", In Charles Rich, Bernhard Nebel, & William Swartout, Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference, Cambridge, MA, Morgan Kaufmann,1992.

[57] Part 2 of the FIPA 97 specifications, Agent Communication Language: http://drogo.cselt.stet.it/fipa/spec/fipa97/f7a12pdf.zip

[58] Homepage of FIPA: http://drogo.cselt.stet.it/fipa/

[59] ARPA Knowledge Sharing Initiative. Specification of the KQML agent communication language. ARPA Knowledge Sharing Initiative, External Interfaces Working Group, July 1993.

[60] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P.Rice. Okbc: A programmatic foundation for knowledge base interoperability. In *Proceedings of the National Conference on Artificial Intelligence (AAAI98*). AAAI Press, 1998.

[61] R. S. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Sobo-roff, J. Mayfield, and A.Boughannam. Jackal: a java-based tool for agent development. In *Working Papers or the AAAI-98 Workshop on Software Tools for Developing Agent*s, august 1998.

[62] M.R. Genesereth. Knowledge interchange format version 3.0 reference manual. Technical Report Logic Group Report Logic-92-1, Stanford University, June 1992.

[63] M. R. Genesereth and S. P. Katchpel. Software agents. *CAC*M, 37(7):48–53, 147, 1994.

[64] O. M. Group. Mobile agent facility (joint submission). Technical report, Object Management Group, 1997.

[65] C. G. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research report, T.J. Watson Research Center, 1994.

[66] Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97*), Nagoya, Japan, August 1997.

[67] 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. *Personal Technologie*s, 2(3), December 1999.

[68] 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 K. Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'9*8, LNCS 1477. Springer-Verlag, 1998.

[69] D. S. Milojicic, M. Condict, F. Reynolds, D. Bolinger, and P. Dale. Mobile objects and agents. In *"Distributed Object Computing on the Internet" Advanced Topics Workshop, Second USENIX Conference on Object Oriented Technologies and Systems (COOTS*), Toronto, Canada, 1996.

[70] H. S. Nwana. Software agents: an overview. *Knowledge Engineering Revie*w, 11(3):1–40, September 1996.

[71] R. S. Patil, R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R.Neches. The darpa knowledge sharing effort: Progress report. In M. Huhns and M.Singh, editors, *Readings in Agent*s. Morgan Kaufmann Publishers, 1997. (reprint of KR-92 paper).

[72] D. Rus, R. Gray, and D. Kotz. Transportable information agents. In *1997 International Conference on Autonomous Agent*s, Marina del Ray, CA, 1997.

[73] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. “Fine-Grained Mobility in the Emerald System”. ACM Transactions on Computer Systems, 6(1): 109—133, February 1988.

[74] James Gosling, Bill Joy, and Guy Steele. “The Java Language Specification”. Addison Wesley, August 1996.

[75] J. Steven Fritzinger and Marianne Mueller. “Java Security. Technical report”, Sun Microsystems, 1996. http://www.javasoft.com/security/whitepaper.ps

[76] Neeran M. Karnik. “Security in Mobile Agent Systems”. PhD thesis, University of Minnesota, October 1998.

[77] Bruce Schneier. “Applied Cryptography”. John Wiley & Sons, 1996.

[78] M. T. Sun and I.-M. Pao, “Multipoint video conferencing,” *Visual Commun. and Image Pro*c., Marcel Dekker, 1997.

[79] T. Magedanz, K. Rothermel, S. Krause: "Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?” in: Proceedings of IEEE INFOCOM '96, pp. 464-472, IEEE Catalog No. 96CB35887, ISBN: 0-8186-7292-7, IEEE Press, 1996.

[80] Crystaliz Inc, General Magic Inc, GMD FOKUS, IBM, TOG: OMG Joint Submission "Mobile Agent System Interoperability Facility", November 1997, available at ftp://ftp.omg.org/pub/docs/orbos/97-10-05.pdf

[81] OMG: The Common Object Request Broker: Architecture and Specification. Technical Report, Revision 2.2, Feb 1998

[82] OMG: Common Facilities rfp3. Request for Proposal OMG TC Document 95-11-3, Object Management Group, Framingham, MA, November 1995.

[83] Grasshopper homepage: http://www.ikv.de/products/grasshopper/

[84] OMG: CORBA services: Common Object Services Specification, November 1997

[85] FIPA: FIPA98 Draft Specification: Part 11: Agent Management Support for Mobility, FIPA8415, Version 0.3, Foundation for Intelligent Physical Agents, April 1998

[86] Sankar Virdhagriswaran http://www.crystaliz.com/logicware/mubot.html - White paper of Crystaliz, Inc., defining mobile agent technology.

[87] Robert S. Gray. Agent Tcl: A transportable agent system. In Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December 1995.

[88] Shankar Sundaram. Mission-Flow Constructor: A Workflow Management System Using Mobile Agents. Masters Thesis, Thayer School of Engineering, Dartmouth College, May 2000.

[89] Lange, D. B. et al., Aglets: Programming Mobile Agents in Java, In *Proc. of Worldwide Computing and Its Applications (WWCA '97),* Lecture Notes in Computer Science, vol. 1274, Springer Verlag, 1997.

[90] Mitsuru Oshima, Guenter Karjoth, and Kouichi Ono Aglets Specification 1.1 Draft, September 8th, 1998.

[91] Odyssey: Beta Release 1.0, 1997. Available as part of the Odyssey package at http://www.genmagic.com/agents/.

BIOGRAPHICAL INFORMATION

Vamsi Putrevu attended The Malaviya Regional Engineering College at Jaipur, India from 1995 through 1999. He started his M.S. in Computer Science and Engineering at the University Of Texas at Arlington, U.S in the fall 1999. He served as a Teaching Assistant of Dr. Mostafa Ghandehari, for the course "Algorithms and Data Structures" in CSE department of UT Arlington from Fall 2000 to the Summer 2001. He received his M.S. in Computer Science and Engineering in August 2001.