INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available.

UMI®

Université d'Ottawa · University of Ottawa

SHIPMAI:
Secure and HIgh Performance Mobile Agent Infrastructure

By

Reda TKITO, B.Eng.

A thesis
submitted to the school of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Applied Sciences

in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering
Department of Electrical and Computer Engineering
Faculty of Engineering

University of Ottawa

April, 2000

© 2000, Reda TKITO, Ottawa, CANADA

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-58513-1

Canadä

ABSTRACT

In the computer area, the term "agent" describes a software entity that performs tasks on behalf of the user. Recently, the mobile agent technology has become an attracting research subject, especially in the field of Distributed Object Technology (DOT), where it is believed that this new paradigm could considerably rectify the shortcomings of its predecessors.

As a relatively new concept, the agent technology is still immature, although it is gathering its share of investment money. In our opinion, a major problem still limiting widespread use of mobile agents is the lack for an appropriate supporting platform. The existing agent infrastructures consider differently the issues related to the paradigm; and none comes with a design that appropriately addresses all of them; security being the most poorly treated aspect.

This thesis presents our Java-based mobile agent platform called SHIPMAI. The proposed platform aims to efficiently provide all necessary agent services, namely, execution, security, transport, tracking, communication, and persistence. Those MA services are reasonably distributed thanks to the smart agent-space hierarchy introduced by SHIPMAI. Our system architecture consists of private agent domains managed by ADCs (Agent Domain Controllers). Each domain contains several AEEs (Agent Execution Environments), which are destined to host agents. The idea is partially inspired from the Intranet concept.

To improve the system flexibility, policies are integrated so as to leverage the functioning of SHIPMAI and to control the behavior of agents. The thesis work includes also the development, on top of SHIPMAI, of a "Mobile Agent-Based Software Distribution" application. MABSD uses agents in order to provide a highly automated delivery and installation of software packages.

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my thesis supervisor, Dr. Ahmed Karmouch, who guides me all along the steps of my project, and who didn't spare any effort to advise me in order to achieve this thesis work.

From our sponsoring company, Nortel Networks, I want to thank Clifford Grossner and Andre Vellino, who were in charge of following the evolution of our SHIPMAI project by attending and evaluating different presentations and demos.

I'd like also to acknowledge all members from the Multimedia & Mobile Agent Research Laboratory for their strong motivation, their friendly work environment, and their interesting discussions and brainstorms.

Special thanks are due to Hamid Harroud, Hassan Houbban, Mouhsine Lakhdissi, and Abdellah Sebbar for their valuable assistance throughout the realization of this work and the writing of this thesis, for their continual encouragement, and for their precious friendship.

I would not forget the administrative staff at the University of Ottawa, who deserves my deep recognition for his wonderful help.

Finally, I want to extend my sincere appreciation to every person who had, in one way or the other, contributed to the accomplishment of this modest work.

Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES IV

LIST OF ABBREVIATIONS V

1. INTRODUCTION 1

1.1. MOTIVATION 1
1.2. OBJECTIVES 2
1.3. MAIN CONTRIBUTION 3
1.4. THESIS OUTLINE 4

2. AGENT TECHNOLOGY 5

2.1. SOFTWARE AGENTS 5
2.1.1. Definitions 5
2.1.2. Characteristics 6
2.2. CLIENT/SERVER PARADIGM 7
2.3. MOBILE AGENT PARADIGM 8
2.3.1. Concept 8
2.3.2. Advantages 8
2.3.3. MA applications 9
2.3.4. Requirements 10
2.4. SURVEY OF EXISTING MA PLATFORMS 11
2.4.1. D'Agents 12
2.4.2. ARA (Agent for Remote Action) 13
2.4.3. Aglets 14
2.4.4. Concordia 15
2.4.5. Odyssey 16
2.4.6. Voyager 17
2.4.7. Grasshopper 18
2.5. CHALLENGING ISSUES 19
2.6. PROPOSED APPROACH 21

3. SHIPMAI DESIGN 22

3.1. INTRODUCTION 22
3.2. MODEL AND ARCHITECTURE 22
3.2.1. Layered architecture 22
3.2.2. User Application Interface (UAI) 23
3.2.3. User Management Service (UMS) 24
3.2.4. Administrative Center (AC) 24
3.2.5. Agent Domain Controller (ADC) 25
3.2.6. Agent Execution Environment (AEE) 26
3.2.7. Domain schema 27
3.3. SHIPMAI AGENT 28
3.3.1. Characteristics 28
3.3.2. Constitution 29
3.3.3. Life cycle 31
3.4. COMMUNICATION 33
3.4.1. Principles 33

3.4.2. Constraints *34*
3.4.3. Message routing *34*
3.4.4. Agent migration *36*
3.5. PERSISTENCE 38
3.5.1. Principles *38*
3.5.2. Agent data *38*
3.5.3. Services *39*
3.5.4. Advantages & Drawbacks *41*
3.6. SECURITY 42
3.6.1. Principles *42*
3.6.2. Analogy with Intranets *43*
3.6.3. Protecting agents *44*
3.6.4. Protecting hosts *45*
3.6.5. Security mechanisms *47*
3.6.6. Advantages & Drawbacks *47*

4. SHIPMAI IMPLEMENTATION 49

4.1 INTRODUCTION 49
4.2. IMPLEMENTATION CHOICES 49
4.2.1. Why JAVA as programmatic language? *49*
4.2.2. Why Directories & LDAP for data management? *51*
4.3. AGENT OBJECT 53
4.3.1. Components *53*
4.3.2. Class diagram *54*
4.4. MESSAGING SYSTEM 55
4.4.1. Components *55*
4.4.2. Techniques & Tools *57*
4.4.3. Class diagram *59*
4.5. SECURITY FRAMEWORK 60
4.5.1. Components *60*
4.5.2. Techniques & Tools *62*
4.5.3. Class diagram *67*
4.6. PROTOTYPE 68
4.6.1. Agent creation *68*
4.6.2. Agent directory storage *69*
4.6.3. Intra-domain & Inter-domain agent migration *70*

5. MABSD APPLICATION 71

5.1 INTRODUCTION 71
5.2. MOTIVATION 71
5.3. OBJECTIVE 72
5.4. OPEN SOFTWARE DESCRIPTION (OSD) 73
5.4.1. Definition *73*
5.4.2. Representation *73*
5.4.3. Specification *74*
5.4.4. Using OSD in MABSD *74*
5.5. ARCHITECTURE 76
5.5.1. System overview *76*
5.5.2. Information Agent *78*
5.5.3. Deliverer Agent *79*
5.5.4. Installer Agent *80*
5.5.5. Directory Server *81*
5.5.6. Graphical User Interface *81*
5.5.7. Scheduler *82*
5.6. CLASS DIAGRAM 83
5.7. MABSD & SHIPMAI 84

5.7.1. Application scenario *84*
5.7.2. SHIPMAI benefits *87*
5.8. PROTOTYPE 88
5.8.1. Package OSD file *88*
5.8.2. Viewing software attributes *89*
5.8.3. Delivery request *90*
5.8.4. Launching delivery process *91*
6. POLICIES-BASED SHIPMAI 92
6.1 INTRODUCTION 92
6.2. BRIEF DEFINITIONS 92
6.3. MOTIVATION 94
6.4. ORIENTATION 95
6.5. POLICIES FOR INFRASTRUCTURE COMPONENTS 95
6.5.1. Architecture *95*
6.5.2. User Management Service (UMS) *96*
6.5.3. ADC and AEE *98*
6.5.4. Administrative Center (AC) *100*
6.6. POLICIES FOR MOBILE AGENTS 101
6.6.1. Vision *101*
6.6.2. Obligation policies *102*
6.6.3. Authorization meta-policies *104*
6.7. POLICIES STORAGE 105
6.7.1. Policies as objects *105*
6.7.2. Infrastructure Components' policies *106*
6.7.3. Mobile Agents' policies *107*
7. CONCLUSIONS 109
7.1. SUMMARY 109
7.2. FUTURE WORK 110
8. REFERENCES 112

LIST OF FIGURES

FIGURE 2.1: CLIENT/SERVER *VS* MOBILE AGENTS 9
FIGURE 3.1: SHIPMAI LAYERED ARCHITECTURE 23
FIGURE 3.2: USER'S REQUEST PROCESSING 24
FIGURE 3.3: SHIPMAI DOMAIN ARCHITECTURE 27
FIGURE 3.4: SHIPMAI AGENT CONSTITUTION 31
FIGURE 3.5: SHIPMAI AGENT LIFE CYCLE 31
FIGURE 3.6: SHIPMAI MESSAGE ROUTING 36
FIGURE 3.7: SHIPMAI SECURITY MODEL 44
FIGURE 4.1: LDAP DIRECTORY TREE EXAMPLE 52
FIGURE 4.2: UML VIEW OF AGENT CLASS DIAGRAM 54
FIGURE 4.3: SHIPMAI MESSAGE TRANSMISSION AND HANDLING 57
FIGURE 4.4: UML VIEW OF THE COMMUNICATION FRAMEWORK CLASS DIAGRAM 59
FIGURE 4.5: UML VIEW OF THE SECURITY FRAMEWORK CLASS DIAGRAM 67
FIGURE 4.6: GUI SCREEN FOR AGENT CREATION REQUEST 68
FIGURE 4.7: GUI SCREEN FOR AGENT DIRECTORY 69
FIGURE 4.8: UML SEQUENCE DIAGRAM FOR INTRA-DOMAIN AGENT MIGRATION 70
FIGURE 4.9: UML SEQUENCE DIAGRAM FOR INTER-DOMAIN AGENT MIGRATION 70
FIGURE 5.1: EXAMPLE OF OSD FILE 75
FIGURE 5.2: MABSD ARCHITECTURE 77
FIGURE 5.3: UML VIEW OF THE MABSD CLASS DIAGRAM 83
FIGURE 5.4: MABSD APPLICATION SCENARIO 84
FIGURE 5.5: INSTALLER_AGENT MIGRATION STEPS 86
FIGURE 5.6: PROTOTYPE PACKAGE OSD FILE 88
FIGURE 5.7: GUI SCREEN FOR "USER / INFORMATION_AGENT" INTERACTION 89
FIGURE 5.8: GUI SCREEN FOR SOFTWARE DELIVERY REQUEST 90
FIGURE 5.9: GUI SCREEN SHOWING DELIVERY LAUNCHING 91
FIGURE 6.1: SHIPMAI POLICIES ASSIGNATION 96
FIGURE 6.2: ADC/AEE POLICY STRATEGY 100
FIGURE 6.3: NATIVE AND VISITING AGENT POLICIES 104
FIGURE 6.4: AGENT POLICY STRUCTURE 105
FIGURE 6.5: DIRECTORY HIERARCHY FOR SHIPMAI POLICIES 107
FIGURE 6.6: AGENT POLICIES CLASS DIAGRAM 108

LIST OF ABBREVIATIONS

AC	Administrative Center
ACL	Agent Control List
ADC	Agent Domain Controller
AEE	Agent Execution Environment
AI	Artificial Intelligence
API	Application Programming Interface
ARA	Agent for Remote Action
ASN.1	Abstract Syntax Notation 1
CA	Certificate Authority
CDF	Channel Definition Format
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRL	Certificate Revocation List
DAE	Distributed Agent Environment
DBMS	DataBase Management System
DCOM	Distributed Component Object Model
DN	Distinguished Name
DOT	Distributed Object Technology
DSA	Digital Signature Algorithm
FIPA	Foundation for Intelligent Physical Agents
FIPS	Federal Information Processing Standards
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
ITU-T	International Telecommunications Union-Telecommunication
JAR	Java ARchive
JCE	Java Cryptography Extension
JDK	Java Development Kit

JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MA	Mobile Agent
MABSD	Mobile Agent-Based Software Distribution
MAF	Mobile Agent Facility
MASIF	Mobile Agent System Interoperability Facility
MMARL	Multimedia & Mobile Agent Research Laboratory
NCL	Network Command Language
NP	Native Policies
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
OSD	Open Software Description
PKCS	Public-Key Cryptography Standards
QoS	Quality of Service
REV	Remote EValuation
RFC	Request For Comments
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman algorithm
SHA	Secure Hash Algorithm
SHIPMAI	Secure and HIgh Performance Mobile Agent Infrastructure
SSL	Secure Socket Layer
Tcl	Tool command language
TCP	Transmission Control Protocol
UAI	User Application Interface
UML	Unified Modeling Language
UMS	User Management Service
URL	Uniform Resource Locator
VP	Visiting Policies
VRMP	Voyager Remote Method Protocol
W3C	WWW Consortium
WWW	World Wide Web
XML	eXtensible Markup Language

Chapter 1

1. INTRODUCTION

1.1. Motivation

Since the beginning of the last decade, the agent paradigm has emerged as a promising design concept and implementation solution for future applications. It gained more popularity in the research community with the explosion of Internet and the actual influence of the World Wide Web (WWW) in the software-programming world. Presently, the mobile agent technology is seen as an attracting key development in the area of distributed systems, a new programming way that could considerably rectify the shortcomings of its predecessors. In fact, it is regarded as the next step in the evolvement of the Distributed Object Technology (DOT).

Mobile agents (MA) are attributed by researchers numerous and different characteristics that make them suitable for a large number of computing tasks; especially those that still need burdening manual intervention. By acting on behalf of users, agents would allow greater automation of tasks in complex software systems. Some interesting agent applications would be related to electronic commerce, software distribution and network management.

However, MA technology has not yet reached a fully mature stage. In fact, up to now, there is no killer application using mobile agents, which are still at the stage of intellectual curiosities. In our opinion, a major problem still limiting the widespread use of mobile agents is the lack of an appropriate and well-designed supporting platform, which efficiently provide all the functionality necessary to the deployment of mobile agents.

Existing MA systems consist mostly of agents migrating between hosting servers dispersed throughout the network. These servers or AEEs (Agent Execution Environments) are equal in the agent architecture. Each server should perform all agent-related functions, including agent execution, control, resource allocation, collection of state information, migration, authentication, etc. This all-in-one AEE design raises two major problems:

The first problem concerns ***performance***: since the AEE provides all agent services, the performance and robustness of the system will be certainly affected as the number of hosted agents grows higher. Moreover, valuable agent work can be easily lost due to AEE crash or network failure. The second problem is related to ***security***: with such design, it is difficult to define an efficient security model. That is, a malicious host may easily harm the whole agent work since it is the only element responsible for its persistence. Also, a malicious agent, if not carefully controlled at each AEE, could damage or mischievously use the server resources.

1.2. Objectives

This thesis has three main objectives. The first one is the development of a Java-based mobile agent platform in order to address especially the two issues of performance and security. Thus we called it SHIPMAI (Secure and HIgh Performance Mobile Agent Infrastructure) [1]. The implemented system consists of building blocks offering the services that are necessary to the deployment of mobile agents. An agent architectural model is also defined.

The second objective is to design and implement an application for automatic software delivery and installation. We baptized it MABSD (Mobile Agent-Based Software Distribution).

MABSD is built on top of SHIPMAI and thus will serve as a validating application to test our infrastructure's ability to support useful applications [2].

The last objective of this thesis is the use of policies to restructure the functioning of the SHIPMAI system. Sets of policies will be assigned to the infrastructure components to regulate their actions based on obligations and authorizations. Policies will be also injected to mobile agent objects in order to ease the change of their behavior.

1.3. Main contribution

The main contribution of this research work is to develop a reliable, easy-to-manage, and highly secure mobile agent infrastructure. The outset of the infrastructure design was specified based on fundamental sponsor requirements. The communication principles were set by the MMARL group researchers. I joined the SHIPMAI project in its early steps and took over the platform development process. The conception work achieved in this thesis includes accomplishing the architecture design, defining an appropriate agent object model, and addressing the persistence and security issues. The implementation work concerned all the system components (agents and servers) and the platform services, namely, persistence, communication, agent mobility, and security.

Another key facet of this thesis consists of restructuring SHIPMAI (agents and infrastructure entities) for a policies-based functioning. Policies, usually used for pure management tasks, are adapted in this work in order to drive servers' tasks and to control completely agents' behavior.

The thesis contribution includes also the development of an agent-based application for automatic software distribution through the Internet (MABSD). The application uses SHIPMAI

as a supporting platform to benefit from its performance and security features, and to validate its appropriateness for promising mobile agent applications.

This research work is part of large collaboration between the university of Ottawa and Nortel Networks Corporation.

1.4. Thesis outline

The rest of the thesis is structured as follows. Chapter 2 introduces software agents. The mobile agent concept is then presented, compared with the Client/Server model, and discussed to show its advantages, requirements and areas of predilection. A subset of existing MA platforms are overviewed and analyzed. Chapter 2 ends by accentuating the challenging issues of the mobile agent paradigm and the approach we propose to address them.

Chapter 3 presents the SHIPMAI design. It begins by describing its architecture and building blocks. The conceived model for agents is then detailed giving the agent objects' composition and life cycle. The platform services are then explained. Stress is put on the security and the persistence aspects. Chapter 4 describes the SHIPMAI implementation, including the agent model, the communication and security frameworks, and the system prototype.

Chapter 5 is devoted to the MABSD application. It identifies its design architecture and presents its implementation prototype. Chapter 6 shows how policies are integrated to SHIPMAI components and agents in order to drive their functional behavior. Chapter 7 summarizes the thesis conclusions and provides some suggestions for future research work.

Chapter 2

2. AGENT TECHNOLOGY

2.1. Software agents

2.1.1. Definitions

The term "agent" is given to a variety of things by a variety of people. A philosophical problem for the agent research community is the extensive use the "agent" term to describe a diversity of software entities. There is no universal and standard definition specifying exactly what a "software agent" is. Researchers have given several definitions, reflecting their own use of the term. Below come some common definitions [3].

- *MuBot Agent*: "The term agent is used to represent two orthogonal concepts: The first is the ability for autonomous execution. The second is the ability to perform domain-oriented reasoning."

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

- *IBM Agent*: "Intelligent agents are software entities carrying out a 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."

- *MMARL Agent*: "A mobile agent is a program that acts on behalf of a user or another program and is able to migrate from host to host on a network under its own control." [4].

2.1.2. Characteristics

At an elementary conceptual level, researchers agree that agents can be regarded as entities performing actions on behalf of the user. However, they differ about the necessary characteristics for the "agent" appellation of a software entity. Depending on their given definitions, agents are believed (or required) to have the following characteristics with various degrees:

- *Autonomy*: maybe the most prominent buzzword. In the sense that, given the bounds and limitations of its tasks, an agent should be able to operate independently from its user.

- *Sociability*: means interacting with the outside world [5]. This interaction can exist at a number of levels, namely with the user, with other agents and with the local environment.

- *Pro-activeness*: helps differentiating between an agent and another piece of software. It is the ability of agents to effect actions to achieve their goals by taking the initiative.

- *Mobility*: it means that an agent is able to transport itself from one machine to another and move across networks to fulfil its goals. This is an interesting feature for distributed applications systems. The mobile agent paradigm will constitute the central topic of this thesis.

- *Rationality*: is the assumption that an agent will not act in a manner that prevents it from achieving its goals and will always attempt to fulfil those goals [6].

- *Benevolence*: it means that an agent cannot have conflicting goals forcing it to transmit false information and does not effect actions that cause its goals to be unfulfilled or impeded [7].

Agents do not constitute a complete application by themselves. Instead, they form one by working in conjunction with an agent host and other agents. In many ways, agents are of the same scope as applets, i.e. of limited functionality on their own.

2.2. Client/Server paradigm

The Client/Server paradigm [8] is well known and widely used in today's computing. Most Networking concepts and techniques, including most Internet protocols, are chiefly based on it. We can also find it in OSs and DBMSs. The server offers centralized services that are processed at its side using local data. The client invokes services or accesses data through the server.

The advantage of the Client/Server model is that it enables the execution of the client in remote smaller machines. Also, the technology is heterogeneous regarding hardware and operating systems; client and servers communicate through standard APIs using RPCs.

However, to be perfectly reliable, the Client/Server paradigm requires high quality in network connections, which is not always available, especially in the Internet. Furthermore, it needs good bandwidth, since due to its very nature, client/server must copy data across the network. This generates great traffic and increases the overall response latency, particularly when large amounts of data are exchanged, i.e. when the client has to make a series of remote calls to obtain the end service desired.

The relatively recent notion of *Subprogramming*, which is a kind of process migration, helps somewhat solving these problems by allowing clients to send their own subprograms to be processed remotely at the server node. This permits the execution of several low-level requests before transmitting results back to the client. Examples of Subprogramming or process migration systems include REV (Remote Evaluation) [9] and NCL (Network Command Language) [10].

While they can migrate on their initial launch, subprograms cannot subsequently move to other systems or resources. Furthermore, they are, in general, explicitly written for specific clients and servers, which considerably minimizes their reusability.

2.3. Mobile Agent paradigm

2.3.1. Concept

Mobile agents are a sub-classification of agents whose predominant feature is the ability to travel between servers across the network to perform tasks. Transportable agents are considered as promising extension of the client/server technology.

A Mobile Agent, besides being an independent program executing on user's behalf, has the ability to travel to multiple locations in the network in order to achieve its mission. Sub-works performed at each node may be independent or related to each other. Mobility greatly enhances the productivity of each software element in the network and creates a uniquely powerful computing environment well suited to a number of applications.

2.3.2. Advantages

It is believed that the MA technology can help considerably resolving problems of distributed computing. It particularly attempts to address the lacks of the Client/Server and Subprogramming paradigms. The characteristics offered by the notions of agency and mobility are found to be beneficial with regard to many aspects [11].

- *Efficiency*: Agents can move to the location where services are offered. They take full advantage of resources local to the many networked machines they visit. They process data at the data source, rather than fetching it remotely. After completely finishing their mission, they return back to the client with the whole work results (see figure 2.1). Such asynchronous behavior reduces network traffic and allows higher performance compared with the synchronous functioning of the Client/Server model. This efficiency is more obvious for distributed applications that involve a great amount of exchanged information.

- *Reliability*: Once migrated, a mobile agent no longer depends on the system node that had launched it, and it will not be affected by its failures. This assures some fault tolerance that is not granted by the client/server architecture. Also, an agent may run even if the user is disconnected from the network, voluntarily or involuntarily. Because the agent data processing takes place locally at the source, the network has no effect on the agent execution.

- *Flexibility*: A drawback of the client/server paradigm is its rigidity in fixing the role of each entity. A mobile agent, because of its sociability and pro-activeness, can play both roles. It is a client when requesting machine resources; and a server when queried by another agent.

Figure 2.1 provides a schematic comparison between the Client/Server and MA paradigms.

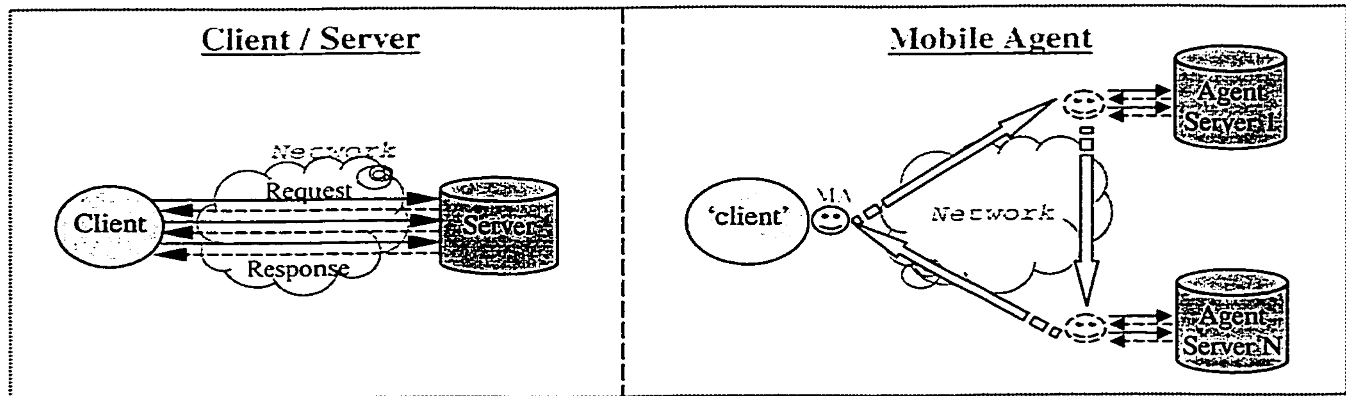


FIGURE 2.1: CLIENT/SERVER *VS* MOBILE AGENTS

On the other side, the MA paradigm presents some difficulties; one of which is the necessity of having agent platform software installed in every hosting machine. Another hardship is security: to what extent can a server trust a foreign program, or can an agent trust a strange host?

2.3.3. MA applications

Domains of predilections for agents are various. Their applications have in common the fact that they are directed toward the Internet and, consequently, are greatly concerned by network bandwidth use. The following are some of the most promising classes of MA applications.

- *Network Management*: MAs are used in network management for different purposes such as collecting state information, invoking changes to switch controllers and managing the network QoS (Quality of Service) [12]. Therefore, there are MA-based applications dealing with route selection [13], local decision making [14] or network mapping [15].

- *Electronic Commerce*: It is a field where MAs have been of substantial use. Many MA platforms such as Magnet [16] have been developed specially to address critical problems in electronic commerce. Actually mobiles agents can carry out a purchasing mission on the behalf of the user according to the specified parameters.

- *Information Retrieval*: This is a branch of artificial intelligence (AI) well suited for MAs. In fact, sending agents to perform local search can be more efficient than looking remotely for specific keywords in distant machines. Potential use of mobile agents for Information retrieval is described in [17].

2.3.4. Requirements

In order to achieve the objective of implementing mobile agent-based applications, an essential and necessary requirement is an appropriate supporting platform. An MA platform should enable the creation, management, and control of mobile agents. This infrastructure is intended to serve as a low-level layer to developed applications. Hence, it should provide all the general-purpose agents' functionality. Fundamental services an MA system must offer include:

- *Communication Services*: Within the platform, mobile agents need to communicate with each other [18]. They also need to interact with the platform components in order to exchange data, intermediate results, and control and error messages. These facilities can be accessed through the communication services.

- *Security Services*: One of the major concerns to the MA research community is the enforcement of adequate security policies regarding the agents and their environments [19]. Agents' access to servers' resources should be carefully controlled. On the other hand, there is a need for protecting agents against malicious hosts. Security must be enforced within a host and while agents are migrating between different nodes in the network.

- *Persistence services*: a primary functionality of a mobile agent platform is to address the problem of persistence. Agents' work and state have to be persistent at any time in order to ensure fault tolerance. This will allow agents to easily recover and retrieve its accumulated work after a technical problem. State persistence also permit agents to resume its execution after an inter-host migration.

2.4. Survey of existing MA platforms

In the last years, intense research efforts were directed towards the design and the implementation of mobile agent systems. This resulted in several academic and industrial products that address quite differently the MA related issues.

In this section, we present an overview of some of the most known MA platforms. Our subset includes ***D'Agents*** (known also as ***Agent Tcl***) from Dartmouth college, ***ARA*** from Kaiserslautern University, ***Aglets*** from IBM, ***Concordia*** from Mitsubishi, ***Odyssey*** (successor of ***Telescript***) from General Magic, ***Voyager*** from ObjectSpace, and ***Grasshopper*** from GMD fOKUS and IKV++.

We will describe briefly each system by showing its important characteristics.

2.4.1. D'Agents

D'Agents, known formerly as Agent Tcl [20] is a mobile-agent system developed at Darmouth college. Agents in this system do not obey to a formally specific model. They are viewed as programs that could be written in different programming languages. The primary interpreted language is Tcl. Lately, support for Java and Scheme was added.

The main component of D'Agents is a server that runs on each machine required to host MAs. It serves as execution environments for agents, which implies that each server should be able to run Tcl scripts (Tcl interpreter), Java programs (JVM), and programs of any language in which system agents can be written. Such requirements can overburden host servers and cause some portability problems.

D'Agents' servers provide also migration facilities for agents via a single instruction. When an agent wants to migrate to a new server, it just calls one function: "*agent-jump*". Migration in Agent Tcl is absolute; i.e. code, state and data are transferred. The state of the agent is completely captured and sent to the destination along with the agent code and data. Once there, the agent is restarted from its last execution point.

Other server's tasks includes keeping track of the agents running on its machine, providing low-level inter-agent communication facilities, basically by message passing. The security strategy is based on authentication, authorization and enforcement [21]. Authentication verifies the identity of an agent's owner (even some servers can accept anonymous agents). Authorization and enforcement control agents' access to server resources and enforce the access limits.

2.4.2. ARA (Agent for Remote Action)

ARA is a mobile agent platform from the University of Kaiserslautern. A common characteristic it shares with D'Agents is that ARA supports agents written in several interpreted languages. "A mobile agent in Ara is a program able to move at its own choice and without interfering with its execution, utilizing various established programming languages" [22].

ARA agents stay (static agents) or move (mobile agents) between servers called ***places***. Places are supposed to offer services agents would use to perform their tasks. Services include, not exhaustively, execution, migration and security. ARA has as design rationale incorporating mobility to the already developed world of programming. "That is, besides mobility there is nothing new to a mobile agent" [23].

Security concerns are basically addressed by restricting the execution environment of the agent. In fact, language interpreters have the property of hiding the host system details by running programs on a virtual machine. Thus agent have no access to the machine processor, memory or operating system. On the other side, there is no general solution for protecting agents against malicious hosts.

When supporting agents programmed in different languages, the system has to deal with portability problems. ARA has the characteristic of sharing the functionality between the interpreter and the core (the appellation given to the run-time system for agents), which both run on top of the same machine operating system. Language-dependent features (e.g. capturing thread execution state) are left to the interpreter to take care of, while language-independent issues (e.g. catching the general state of agents) are the responsibilities of the core.

2.4.3. Aglets

Another existing MA system is Aglets [24] from IBM. Agents in this platform have the same functioning principle as Java applets, thus their "Aglets" appellation. Aglets are mobile Java objects that roam the Internet from one host (called aglet context) to another. In migration, the aglet code and state are transferred using Java's object serialization. The execution state is not captured at the thread-level, so the programmer has to implement its own flow control to define agent execution checkpoints.

All aglets are programmed by inheriting from the same predefined framework (the *Aglet* class), and overriding its methods to add application-specific functionality. Mother class attributes include a global unique identifier and an itinerary object to every aglet. At the occurrence of specific events (such as the reception of an agent), Aglet contexts invoke specific methods (with predefined names) on agents. This reduces the flexibility of the platform.

As a security provision to protect hosts, the system differentiates between trusted and untrusted aglets [25]. The decision to trust or not trust an aglet is entirely up to the host. If the aglet is trusted, the security manager can extend its access rights to real machine resources such as network or file system.

Aglets' communication is based on a message passing mechanism. Proxies are used to relay message objects to and from agents. A proxy is an object representing a specific aglet. It serves as a shield that protects the aglet from direct access to its public methods. This has the advantage of providing location transparency. On the other hand, it may cause some management and performance problems because of the required high mobility of agents.

2.4.4. Concordia

Concordia [26] is an MA infrastructure developed by Mitsubishi Electric. Concordia is implemented using Java. This ensures its portability and cross-platform independence. A Concordia agent is a collection of Java objects encapsulating its code and data. The agent itinerary is described in an independent object stored outside the agent. It contains the destinations' addresses and the entry agent method that should be invoked at each node.

The Concordia infrastructure is composed, like most other MA platforms, of environments destined to host agents. Each hosting node has the heavy burden of providing all required agent functionality. Hence, it consists of several interacting server components that run on one or more JVMs. Particularly, agent mobility or transfer in Concordia is the responsibility of the so-called Conduit server. An agent launches its migration process by invoking methods of the Conduit Server, which then propagate the agent to the Conduit Server of the destination node.

The Concordia communication framework provides two forms of agent interactions. The first is *asynchronous distributed events.* Agents register for events in an Event Manager, which is also in charge of receiving and dispatching events to interested agents. The second form is *agent collaboration* [27]. It allows group communication and requires the specification of "*AgentGroup*" objects that implement collaboration via distributed synchronization points.

Concordia provides fault tolerance via a Persistent Store Manager. Agent state persistence helps successful recovery of agent after a system crash. Storage protection is handled by encryption while host resources protection is based on identities and passwords of users (agent owners). Such mechanism is not scalable because each server must have a global file containing all users' passwords.

2.4.5. Odyssey

Initially, General Magic developed Telescript [28], which proposes its own object-oriented language for programming agents. The Telescript language has the advantage of catching the agent state of execution at the thread level. However, mainly because it obliges agent developers to learn a totally new language, Telescript failed as a commercial product. It was replaced by a Java-based system that was given the name Odyssey. Unlike Telescript, Java does not allow the capture of thread's execution state.

Odyssey inherited most of the general features of its predecessor except for the programming language. Agent servers, called places, are Java programs running on different OSs. They are abstractions of places where an agent can live and carry out its tasks. The two high-level classes for Odyssey agents are *Agent* and *Worker*. *Agent* is the platform abstract class. *Worker* inherits from *Agent* and implements concrete agents performing specific work.

The platform security is based on *authorities*. Each system entity (agent and place) is being taken in charge by an *authority*. By querying the agent authority, a place will accept or reject the agent. It will also issue a permit containing the access rights and limits the agent should respect; otherwise, It will be terminated. It is worth noting that as the number of agents grow, authorities themselves are hard to manage.

In Odyssey, agents use low-level communication by invoking each other methods. The platform offers also event-signaling facility. A design characteristic of Odyssey is its accommodation for different transport protocols including IIOP. Nonetheless, very little public documentation about the Odyssey product is available.

2.4.6. Voyager

Java-Based MA paltforms are currently the norm. ObjectSpace follows suit by developing their Voyager product [29]. Among other agent systems, Voyager is the most tightly integrated to the world of Java object-oriented programming. It is even proclaimed as an agent-enhanced Java ORB (Object Request Broker).

In Voyager as well, agents are programmed by inheriting from a top *Agent* class, and overriding it for specific needs. Agents are assigned global unique identifiers, which particularly serve to track down agents via a name service. Agent migration is held using a mechanism based on virtual classes. A virtual class is equivalent to a Voyager agent Java class with the property of being remotely accessible. Hence, an agent can be accessed by instantiation of the agent virtual class at the remote destination host. The hosts possess thus virtual references that provide access transparency for agent objects.

The system offers advanced communication mechanisms that benefit from the Voyager ORB features. Agents can communicate by invoking each other methods on virtual references. Voyager has its own native protocol called VRMP (Voyager Remote Method Protocol) [30], but it currently supports CORBA, RMI and COM/DCOM as well. The Voyager universal messaging layer allows synchronous, one-way and future-reply invocations on agent objects. Multicasting is also available.

Security, like in the majority of other MA platforms, is poorly treated in Voyager. The infrastructure includes a simple and lightweight security framework that is not particularly adapted for the MA paradigm; its implementation is entirely based on industrial standards. Also, resource management is completely left to Java to take care of.

2.4.7. Grasshopper

IKV++ joins the agent research community by creating its Grasshopper infrastructure for mobile and stationary agents [31]. Grasshopper claimed to be the first agent platform that is compliant with the OMG MASIF standard [32]. It was lately enhanced with an add-on in order to be FIPA conformant [33] and to support mobile and intelligent agents. The system is completely implemented in Java, and thus can run on any machine supporting JVM.

Grasshopper is considered as a universal middleware platform achieving integration between the Client/Server model and the mobile agent paradigm. Therefore, it tries to use the distributed computing technologies (Java and CORBA) to implement a flexible infrastructure. Grasshopper builds a Distributed Agent Environment (DAE) where the agents live and interact. The DAE consists of regions, places and agencies [34]. Even in this subdivision, almost all agent and management services are provided by agencies, with places providing only logical grouping of agents. A region acts mainly as a registry for the other platform entities.

A communication service is in charge of handling all interactions that take place in the DAE. The service uses CORBA and Java RMI and supports both synchronous and asynchronous communications. Each agency has a registration service listing all the agents and places currently hosted. This service is connected to the region registry for tracking purposes.

Grasshopper implements also a security service offering two types of security mechanisms: internal and external. External security deals with interactions between the platform server components e.g. information exchange of agent migration; it is mainly based on authentication and encryption. Internal security concerns the agency's resources protection; its implementation is totally based on JDK security package.

2.5. Challenging issues

MA infrastructures are fundamental for the deployment of the mobile agent paradigm. Hence, the agent technology would not reach a wide popularization unless MA platforms address adequately certain challenging issues.

Mobile agents has not yet make their way through the software computing industry as efficiently as other technologies such as client servers. This particularly means that existing mobile agent platforms (those we discussed in this chapter and others we did not) still do not treat appropriately all the facets of MA requirements.

The first issue to deal with is portability. In a distributed computing world where heterogeneity is an unalterable fashion, every network software system, aiming to keep chances for success, should be designed to run on different platforms. MA systems are not exceptions. The appearance of JVM makes it easier to resolve this problem, thereby letting designers focus on more agent specific issues. That is why most of agent infrastructures opted for Java in their implementation. On the other hand, platforms like ARA and D'Agents, implemented in different scripting languages and depending on OS specific cores, are certainly to present problems of portability and scalability.

The next issue is communication. It can actually be seen as an aspect that covers many others. In fact, communication should not be considered only as interactions between different agent system entities (i.e. agents and infrastructure components). It may include agent mobility, localization and control as well. Most existing MA systems differentiate considerably between these services in their design and implementation. A uniform communication framework will

probably be a better strategy. It would avoid useless effort redundancy when designing different techniques for each service, would be less complex to implement and simpler to manage.

The security aspect is a very crucial issue. Paradoxically, it is poorly treated by almost all MA platforms. Without great concern about security, the mobile agent paradigm will never hit its promising success. Security in MA platforms concerns all other services, especially communication. Since the use of Internet is inevitable, strong security provisions should be taken to secure communication messages and agent migrations through such an unreliable network. The risks of passive and active attacks are enormous. Standard security technologies are necessary, but far from sufficient in the case of mobile agents. Parallel measures should help. For example, a persistence service must be available to ensure recovery of agent data and state if attacked or lost.

Two other very important security matters are the protection of agents against hostile execution environments (host servers), and the protection of hosts against malicious agents. Again, roaming a network like Internet, agents are to visit and be executed by servers they may know nothing about. Similarly for servers that would have to run completely foreign software entities and give them access to system resources. Agents and hosts are, for most cases, far away from trusting each other. Current agent platforms address security concerns nearly in the same manner, which consists mainly on leaving them to the care of the implementation language. Java surely provide efficient security measures, especially with its sandbox model for resource access control. Nonetheless, Security in MA systems is too complicated to be handled uniquely by a programming language. The wrong of designers of MA infrastructures is to often consider security as a system add-on that would be thought of the end of conception cycle, or even until the coding phase.

2.6. Proposed approach

As announced in the introduction, we developed our own mobile agent platform called SHIPMAI. Our infrastructure attempts to overcome the shortcomings that are present in the existing MA infrastructures and to address more appropriately the aforementioned challenging issues that face the expansion of the MA paradigm. Particularly, we have a major concern about the security aspect, which influences greatly our platform design.

When analyzing current MA platforms, we observed that all these systems share a common property: They all mainly consist on mobile agents and machines serving as hosting environments for agents. Each agent server in these infrastructures is required to have all features and to offer all services necessary to the deployment of agents. This could affect gravely the system performance and reliability in case of server crashes for example. Also, in such all-in-one design, agents have to undergo the whole security and checking measures at each node of their itinerary. This makes them redundantly lose valuable time that they could more efficiently use to perform their application tasks.

We believe that an intelligent hierarchy in the agent space, and a reasonable distribution of MA services, would certainly lead to better control and management of agents and services. Therefore, SHIPMAI introduces a subdivision of the network into private agent domains under the authority of ADC servers (Agent Domain Controllers). Each domain will shelter several AEEs and will be controlled by one ADC. We are also somewhat motivated by an analogy with the widely successful concept of Intranets. We can laxly associate Intranets with domains, local servers with AEEs and firewalls/proxies with ADCs. With such analogy, interesting concepts for agents' security and control could then emerge.

Chapter 3

3. SHIPMAI DESIGN

3.1. Introduction

This chapter is devoted to presenting in detail the SHIPMAI design. The design process of our infrastructure had began few time before I get involved in the SHIPMAI project. The idea of a hierarchical architecture was part of the requirements of the industrial sponsor. The first bricks built in the project consisted of a general conception of the overall architecture [35], and a proposition of the communication principles [36]. A part of my contribution was to complete the platform architecture design, to make up a clear and well-defined agent model, and to address other issues that are of great importance to the MA paradigm, such as persistence and security.

We will begin this chapter by presenting our system's architecture, and the role of its components. This will be followed by a section describing the SHIPMAI agent model. We will then explain briefly the communication infrastructure. The two last sections will be dedicated to the SHIPMAI conception regarding, respectively, the persistence and the security issues.

3.2. Model and architecture

3.2.1. Layered architecture

Since the early design steps, SHIPMAI was based on a dual design consisting of two main components that are the ADC (Agent Domain Controller) and the AEE (Agent Executive

Environment). However, these only two elements cannot make a complete and useful infrastructure if not coexisting with other necessary components. That is why we had to add three other components for users-related and administrative purposes. These are the *User Application Interface* (UAI), the *User Management Service* (UMS) and the *Administrative Center* (AC). The five components are parts of a layered architecture that connects them and regulates their different interactions (figure 3.1).

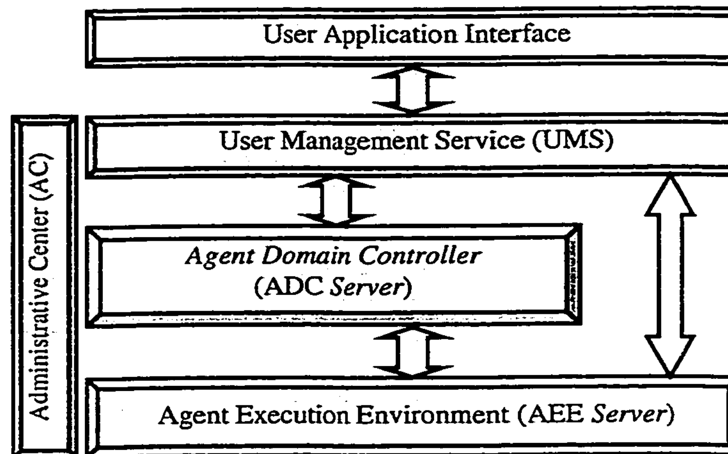


FIGURE 3.1: SHIPMAI LAYERED ARCHITECTURE

3.2.2. User Application Interface (UAI)

This is the interface of the system with final users, which are generally agent owners requesting services from their agents or from the system. SHIPMAI offers a graphical user interface (GUI) that allows users to intervene and interact with the platform in order to:

- Create new agents and assign missions or task lists to them.
- Add/remove jobs to/from agents' task lists and get results of their work.
- Ask for agents' current locations (the tracking mechanism will be briefly described later).
- Kill or terminate agents at completion of their mission, or even prematurely.

3.2.3. User Management Service (UMS)

It is the layer component under the UAI. It is responsible for the management of users. This includes their registration (storing their information), authentication (when accessing the system). It is also the service that receives the user's requests from the UAI and then:

- Checks their legitimacy, e.g. sees if the request is allowed by the system policy, and verifies that the targeted agent belongs to ACL (Agent Control List) of the user.
- Forwards the request to the adequate platform entity that would handle it.

The processing of a user request is shown in the following figure (figure 3.2)

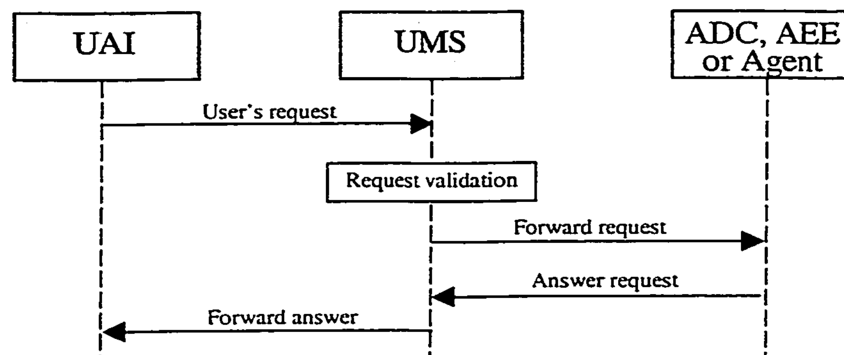


FIGURE 3.2: USER'S REQUEST PROCESSING

3.2.4. Administrative Centre (AC)

By analogy with Intranets, the AC plays the role of the system administrator. It consists of a set of applications for administrating the infrastructure servers and the software components of the domain (users being managed by the UMS). The Administrative centre has full control of all servers and directories and guarantees the domain welfare. Its tasks include the following:

- Starting, restarting and initializing domain servers (ADC, AEEs, ...).
- Doing backups for the system's critical data.

- Applying, changing and controlling the system's policies regarding the access control to domain resources. Further work concerning the integration of policies in SHIPMAI will assign the AC the task of managing domain policies by means of meta-policies. (See chapter 6).

3.2.5. Agent Domain Controller (ADC)

The ADC constitutes a valuable piece in the design of our infrastructure. It centralizes the management burdens, letting AEEs focus on better hosting agents to perform their duties. The tasks of an ADC can be classified under three broad categories.

3.2.5.1. Agent & AEE management

In few words, the ADC should be aware of everything and must have control over all the entities (AEEs and agents) that exist in its domain. Its management work includes:

- Registering of AEEs that are to be added to the domain.
- Registering new agents, storing their attributes, and generating certificates for them.
- Controlling agent transfers based on the resources available in AEEs.
- Serving as agents' lookup directory by continual tracking of domain native agents.

3.2.5.2. Central persistence

An important feature of the ADC is the fact that it acts as central repository where agents regularly send their work and state. During migration, the agent carries only the work results that it would need during its execution in the next hop. This is beneficial with regard to two aspects:

- Enhancing the robustness of the system by providing a central fault-tolerance.
- Reducing agent cargo during migration, thus reducing network bandwidth use.

Of course, on the other hand, sending work results to the ADC generates extra network traffic.

Nonetheless, for MA applications of predilection, the importance, the criticality, and the amount of the gathered information justify its regular transmittal to the ADC separately from the agent.

3.2.5.3. Domain security

The most important preoccupation of the ADC is to guarantee the security of its domain and agents. The notion of privacy introduced by SHIPMAI domains changes the reflection about security in mobile agent platforms. Playing the role of an Intranet firewall, the ADC has complete control over agents and messages entering and leaving the domain. A more detailed discussion about security deserves its own section later in this chapter.

3.2.6. Agent Execution Environment (AEE)

AEEs are the operating environments where agents are executed to perform their tasks. In SHIPMAI, an AEE accepts its domain ADC as the underlying authority. The role of an AEE is significantly reduced comparing with other MA platforms, and consists of the following.

3.2.6.1 Resource allocation

The AEE is responsible for allocating the necessary resources to incoming agents in order to fulfill their missions. This allocation is subject to system policies and ADC directives. As an application level option, AEEs can offer interfaces to their available services and resources, so that they can be discovered by users or even by agents themselves [37].

3.2.6.2. Agent safe execution

Mobile agents cannot be executed by themselves. AEE are then asked to initiate and control their execution. Besides strict security measures applied by the ADC, AEEs can perform some minor security measures such as certificate checking and signature verification. On the other hand, ADCs can limit the damage of malicious AEEs by forbidding them from hosting agents.

3.2.6.3. Migration and communication facilities

Mobile agents cannot migrate by themselves. Hence, AEEs are responsible for transferring them to their desired destination and inform ADC about their movements for tracking purposes.

Mobile agents are by nature difficult to reach because they do not have a permanent address. For this reason, SHIPMAI AEEs supply active agents with mailboxes. Messages destined to an agent will be sent to his host AEE that puts them in the agent mailbox. Also, an agent gives its outgoing messages to the AEE, which will be in charge of sending them to destination.

3.2.7. Domain schema

The following graphical view (figure 3.3) expresses visually our philosophy for the SHIPMAI platform, and summarizes the peer interactions between the different infrastructure entities present inside a domain.

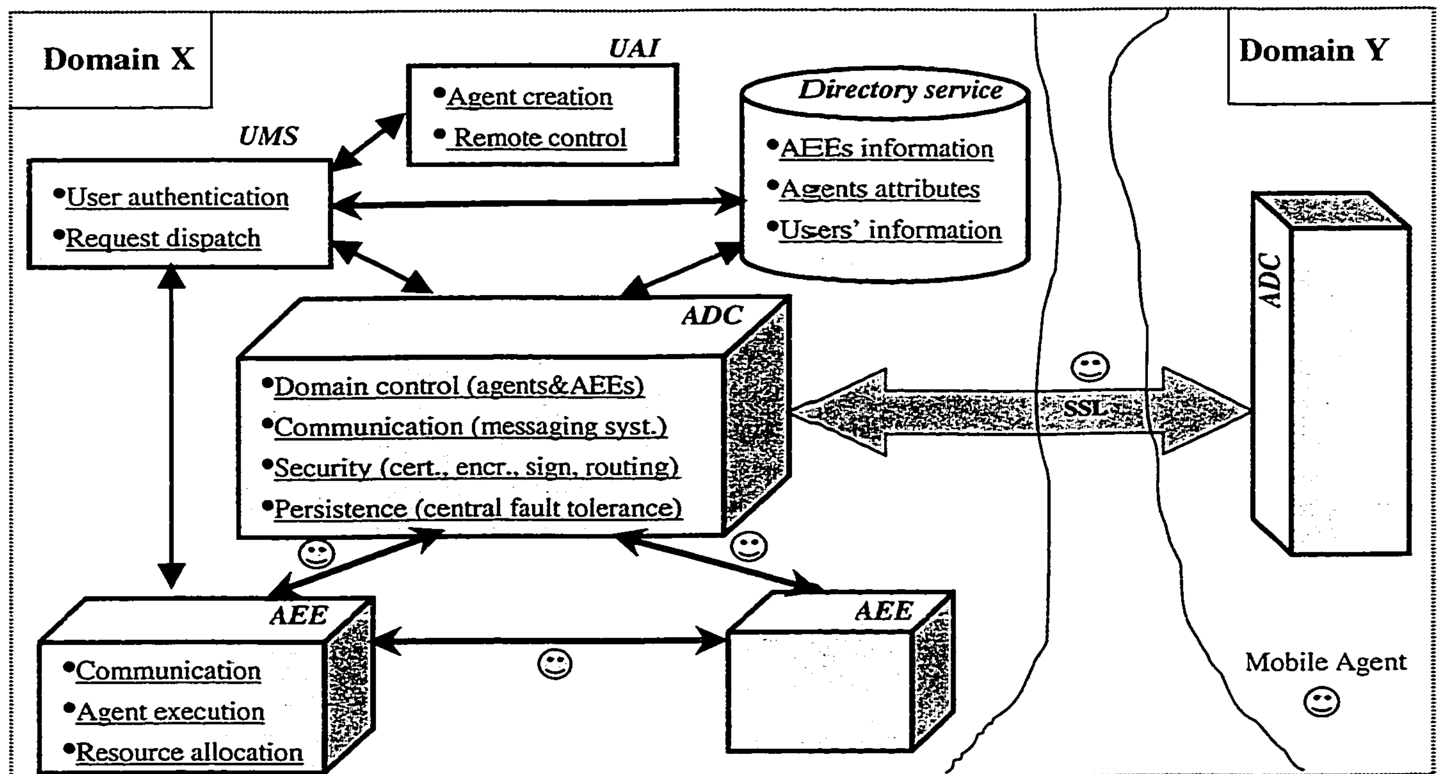


FIGURE 3.3: SHIPMAI DOMAIN ARCHITECTURE

3.3. SHIPMAI agent

3.3.1. Characteristics

As noted in the previous chapter, agents, according to their multiple definitions, are required to have some characteristics with different degrees. In our research lab, a mobile agent is defined as a program that acts on behalf of a user or another program and is able to migrate from host to host on a network under its own control [4].

In SHIPMAI, agents derive their most important features from this definition. Thus, a SHIPMAI agent has the following characteristics:

3.3.1.1. Mobile/Static

Although SHIPMAI is primarily designed for the deployment of mobile agents, it supports static ones as well. The latter differ only by the fact that they do not migrate between different servers, and are rather hosted and executed by a single AEE. Static agents are intended to satisfy user's requests or cooperate with incoming mobile agents to facilitate their mission.

Mobility for other agents in SHIPMAI does not mean the ability to move by themselves, but the ability to decide when and where to migrate. The itinerary of an agent could be specified at the moment of creation and mission assignment or it can be updated by remote control through messages. Also, depending on the application, an agent can choose a destination by checking a service directory. As for the agent transport, it is a service offered by the AEE as part of the communication facilities.

3.3.1.2. Lightweight

When we say mobility, we think systematically of network traffic. Hence, from the performance perspective, an agent should be enough weightless to reduce the bandwidth use

during its travel. The problem is that, generally, mobile agents get gradually heavier by accumulating work results. In our platform, we try to keep agents as light as possible during migration. A SHIPMAI agent does not carry its whole work with it when travelling from one node to another. Instead, it sends all its accumulated results to a central work depository located in the ADC, and keeps only a copy of results it will need in the next hop. Besides ensuring tolerance to agent work loss, such strategy lessens the latency of agent transfer and the consumption of network resources.

3.3.1.3. Reactive

Another important characteristic of agents in SHIPMAI is the fact that they are reactive. Reactivity is the ability to respond to external events and to change behavior according to them. The most common and concrete form of this characteristic in SHIPMAI is the reaction of agents to their incoming messages. Such messages could be related to the application the agent is intended for. But in the infrastructure scope, what we mean by being reactive is the aptitude of agents to respond to directives and events originating from platform entities, i.e. ADC, AEE or other agents. For example, an agent should send its work when requested to do so by the ADC, or ask the AEE when it is short of resources.

3.3.2. Constitution

Among other differences, agents differ from simple software programs by not being constituted of runnable code only. In SHIPMAI, an agent has four different components: identity, platform-dependant code, Application-dependant code and a fourth component for describing its execution state and storing its results.

3.3.2.1. Identity

It consists of immutable data specifying agent attributes such as its name, its owner, its home ADC, its certificate, its mission nature, etc. It gives different sort of information about the agent and allows it to be described, tracked and authenticated.

3.3.2.2. Platform-dependant code

It is the set of programs that enable the agent to interact with other entities of the infrastructure. This code is common to all SHIPMAI agents since it is independent from their specific applications. Using the object-oriented vocabulary, we will say that this part consists of methods that will be inherited by agents at the moment of creation.

3.3.2.3. Application-dependant code

This is the common part between agents and traditional software programs. It is the code allowing agents to perform their missions in the context of specific applications; e.g. electronic commerce, software distribution, or information extraction. It is up to the agent owner or designer to implement this part before creating the SHIPMAI agent.

3.3.2.4. State & Results

An agent performs in an asynchronous manner, thus, it needs a space where to store the results of its processing before returning back to the user, or before sending them to the ADC. Likewise, since a mobile agent may stop its action and migrate to another host to continue its work, it should know the state of its execution. This state is described in a special attribute object so that the agent can resume its task from the last execution checkpoint it reached before migration. The execution state contains the name of the method that should be executed after the hop.

Figure 3.4 represents a SHIPMAI agent with its different components.

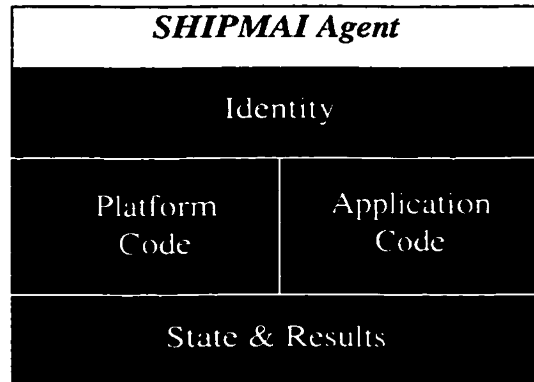


FIGURE 3.4: SHIPMAI AGENT CONSTITUTION

3.3.3. Life cycle

3.3.3.1. Diagram

In this subsection, we will go through the different phases of the life cycle of a SHIPMAI agent (see figure 3.5). We will describe the overall processing in each step. Details about communication and security measures involved in each phase or state transition can be found in the subsections devoted to these issues.

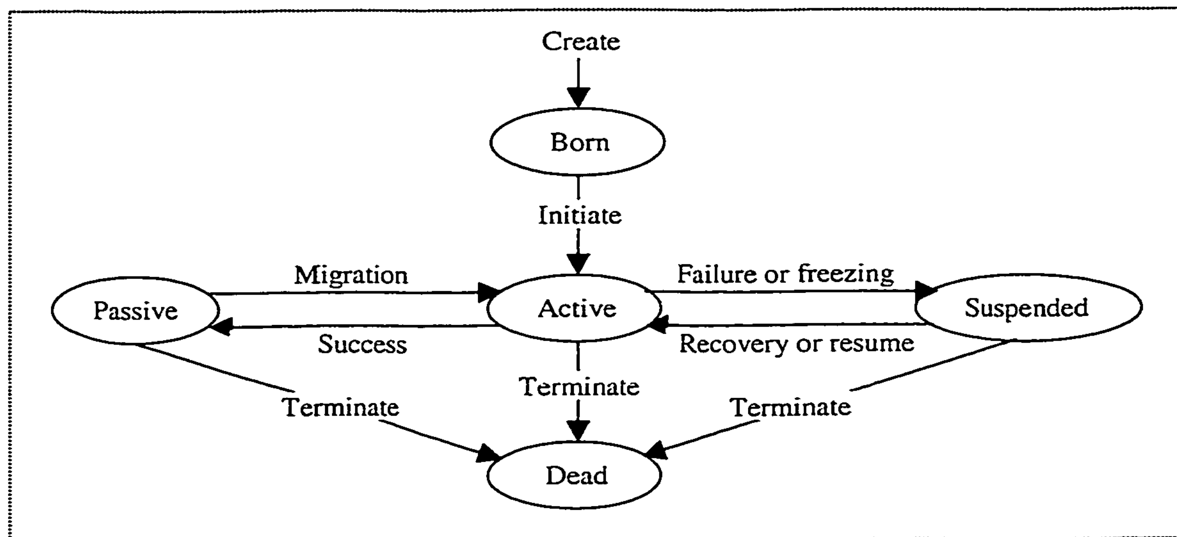


FIGURE 3.5: SHIPMAI AGENT LIFE CYCLE

3.3.3.2. Creation & Registration

To create an agent, a user should first submit a request to the user management service (UMS) of the domain. After authenticating the user, the UMS forwards the request to the ADC, which analyses the request and verify whether it is conform to the domain policy. After that, the user is asked to fill out an agent identity form and to supply application code files. The system then creates the agent object, which includes the platform-dependant code as well.

Comes after that the registration phase: the agent is sent to the ADC which

- registers it to the domain agents' directory,
- stores copies of its attributes and code (so that it can be restituted after loss or tamper),
- generates and attaches a digital certificate to it, and
- initializes it and transfers it to the first host in its itinerary.

3.3.3.3. Execution & Migration

When received by a host environment, the agent is executed by the AEE. This is the phase where the agent is active, performing its tasks and interacting with other entities (e.g. carrying out directives from the ADC, communicating with other agents, etc).

During its execution, an agent might want to migrate to another host, either under its own initiative or due to an external directive or event. In this case, it is brought to a suspended state and transferred to the next destination where it will resume its execution.

In SHIPMAI, there are two types of agent migrations (intra-domain and inter-domain). They are explained in the communication section.

3.3.3.4. Freezing & Termination

Freezing an agent means bringing it to a suspended state while keeping it in the same host. This is done by the AEE in case it is short of resources required by the agent, or when the agent finished its assigned task and is waiting for instructions from its owner or from the ADC.

Freezing keeps the agent object alive, its attributes stored, and its subscription maintained. On the other hand, the termination of an agent destroys all its data, erases all its attributes from the directory server, and unsubscribes it from the domain register. A terminated agent is a dead agent: it can never be brought back to life. An agent termination may occur:

- If the agent completed its mission and gave back the results of its work.
- Upon a termination request from the agent's owner or from the ADC.
- If the AEE noticed a harmful behavior of the agent.

3.4. Communication

3.4.1. Principles

In a mobile agent platform, an adequate communication infrastructure is needed for agents to communicate, and for system components to interact with each other and to control agents. The design of the SHIPMAI communication framework [36] aims to satisfy three principles or goals, which are uniformity, simplicity and flexibility.

The whole communication mechanism of SHIPMAI is built on the top of a messaging model. This means that every peer interaction, every service request and every agent migration is performed by means of transmission and reception of messages.

3.4.2. Constraints

The SHIPMAI communication has two main constraints or preoccupations:

3.4.2.1. Fault tolerance

The results accumulated by the agent and its execution state should be communicated to the ADC before each migration. This ensures fault tolerance and easy recovering, but implies some communication overhead. Also, the large control of ADCs over their domains restrains the freedom of communication between entities belonging to different domains.

3.4.2.2. Security

For the MA paradigm, security is very important and worthily implies special provisions. SHIPMAI, with its notion of private domains, has its own communication constraints. The most important of which is that every message entering or leaving the domain should transit via the ADC to perform security operations. This will highly influence the message routing that will be discussed in the following section.

3.4.3. Message routing

We mentioned that the SHIPMAI communication mechanism is based on the use of messages. Let us see how these messages are routed between different infrastructure entities, and what are the factors ruling this routing.

3.4.3.1. Hierarchy influence

In SHIPMAI, the message routing mechanism is mainly influenced by the platform hierarchy. AEEs are only free to communicate directly with entities located in the same domain. When an AEE wants to communicate with another entity in a foreign domain, it cannot contact it directly even if it knows its exact location (e.g. its IP address and port number). Instead, its sent

and received messages should transit via involved ADCs. This is why the routing in SHIPMAI is handled at a high level.

3.4.3.2. Message handlers

In SHIPMAI's messaging model, ADCs and AEEs have one or more associated queues that play the role of message handlers. The ADC has two queues: an "internal" one for messages from AEEs and agents inside its domain; and an "external" one for messages from foreign ADCs. Also, being a living habitat for active agents, an AEE has three queues:

- One for its own incoming messages.
- One for messages destined to the agents it currently hosts.
- One where hosted agents put their outgoing messages for transmission.

A message handler is responsible for the treatment of incoming messages. This treatment in the SHIPMAI consists of invoking appropriate methods of interested objects. (See paragraph 4.4.2.1. in chapter 4). Those objects should formerly register to the message handler and subscribe for the type of messages they are interested in.

3.4.3.3. Agent mailboxes

Because of their mobility, agents do not have a fixed location where they can be directly reached. For this reason, whenever an agent is given access to an operating environment, the AEE assign it a mailbox for its correspondence. Agents check regularly their mailboxes and react accordingly to received messages.

When an agent leaves the AEE, its mailbox is not destroyed until it has been assigned a new one in the next AEE. Such arrangement is possible thanks to the ADC, which is continually aware of the movements of agents.

3.4.3.4. Routing example

Let us suppose that an agent "a" in a domain X wants to send a message to an agent "b" in a domain Y. the routing scenario will be the following:

- Agent "a" will put the message into the AEE queue for agents' outgoing messages.
- From the identity of the recipient, the AEE will know that it is located in a foreign domain (see implementation of "Identity") and so it will forward the message to the ADC.
- The ADC will pop the message from its "internal" queue, and send it to the ADC of the destination domain, which will receive it in its "external" queue.
- The destination ADC will forward the message to the AEE hosting the agent "b".
- The AEE will put the message into agent "b" mailbox. (see figure 3.6).

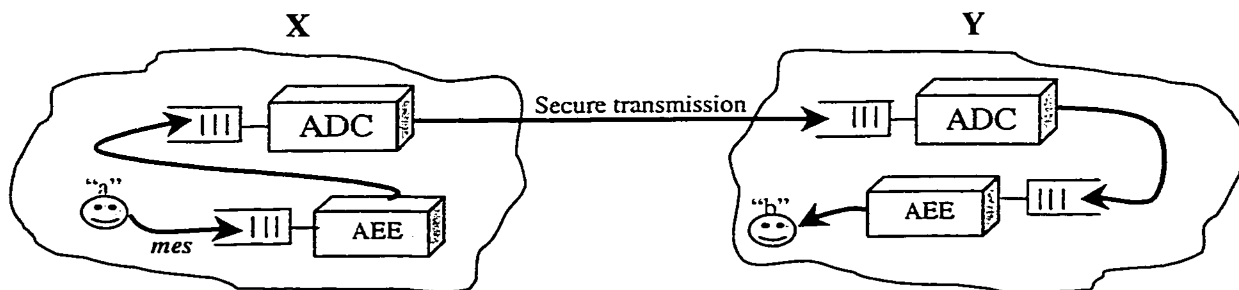


FIGURE 3.6: SHIPMAI MESSAGE ROUTING

3.4.4. Agent migration

In simple terms, we can say that an agent migration in SHIPMAI has the same principle of a message transmission. However, an agent transfer implies a more complex processing than the

transmission of a simple message. In SHIPMAI, we differentiate between two types of agent migration: the *intra-domain migration* and the *inter-domain migration*.

3.4.4.1. Intra-domain agent migration

When an agent wants to migrate from one AEE to another in the same domain, the majority of interactions are done between the two AEEs with a light involvement of the ADC.

After taking the authorization from the ADC, the initial AEE enters in a phase of resource negotiation with the destination AEE. Then, the agent is enveloped into a message object (see chapter 4 for implementation details) and sent directly to its new executing environment. When the destination host receives the agent, it informs the ADC which updates the agent location. The class files then loaded and the agent is executed to accomplish its duty.

3.4.4.2. Inter-domain agent migration

In this case, the onus is on ADCs to transfer the agent between the two domains involved. As in the previous case, the agent request is forwarded to the ADC. The controller enters in interaction with its counterpart ADC to negotiate the resources allocation and the security requirements for the agent. If the two controllers reach an agreement, the home ADC receives the migration message containing the agent from the AEE. Then the two ADCs establish a secure connection to transfer the agent. The foreign ADC then forwards the agent to its desired location, receives an acknowledgement from the AEE, and informs the home ADC about the agent's new location to update it. The agent is then executed in the new AEE.

3.5. Persistence

3.5.1. Principles

Agent data management is an important issue that should be well addressed by a MA platform. Of course, a mobile agent has the ability to carry its work cargo with it, but we think that agents themselves should be persistent. In SHIPMAI, we use a directory service for persistent storage.

As detailed in what follows, the ADC plays the most important role in the persistence strategy. It centralizes all information and offers services that are closely related to the persistence facility.

3.5.2. Agent data

As discussed in the "SHIPMAI agent" section, an agent in our infrastructure is composed of different parts. Some are set at the moment of creation and are practically unchanged (Identity and Codes). Others are often updated and changed (Work and State).

3.5.2.1. Identity & Code

For these immutable components, a permanent persistence is ensured at the ADC level. At the registration of the agent, the controller stores these pieces of data and does not delete them unless the agent is terminated.

Thus, at the difference of most other platforms where the agent exists only in the runtime, SHIPMAI provides a physical and persistent storage of the agent, even for its program codes. At the expense of some disc space, such storage allows a quicker agent re-initialization without going again through the steps of creation and registration.

3.5.2.2. Work & State

Maintaining this data persistent requires more management since it is updated frequently. Information describing agent's work and state is continuously updated in the space allocated to them in the agent object. But for performance purposes, only at regular time intervals or upon agent migration, such data is to be sent to the ADC. Also at the controller level, this information is cached in runtime variables and is persistently stored at larger time intervals. The caching would permit easy recovery in case of agent problem, and the persistent storage would permit it in case of an ADC failure, which should be, normally, scarcely to happen.

3.5.3. Services

Besides ensuring fault tolerance, the persistence strategy of SHIPMAI facilitates the setting of different services related to the use and control of agents. Below, we describe two services that are implemented in our infrastructure. These are agent cloning and agent tracking.

3.5.3.1. Agent cloning

Cloning an agent means creating an identical copy of it. An agent and its clone will:

- Have the same program codes.
- Share the majority of identity attributes except the name and the certificate.
- Have the same accumulated work and execution state at the moment of cloning. After that, each one will have its own life, and certainly these attributes will differ.

In SHIPMAI, thanks to the persistence strategy, an agent can be very easily cloned by the ADC. This latter has all of the data necessary to built up an agent with the same goals, the same

attributes, the same execution state, and to supply it with the same work results so far accumulated by the original agent. Cloning agents is interesting in the following situations:

- The same work is to be done in many hosts. In this case, multiple clones could be created and each one sent to a different node to perform the work. (e.g. in case of an application for information extraction).

- Two or more agent tasks are mutually independent, and can be achieved in parallel.

- If an agent has been corrupted in a host or lost during migration, another agent can be cloned and asked to continue the mission.

3.5.3.2. Agent tracking

Another service that uses the persistence facilities offered by SHIPMAI is the agent tracking. We will just skim through the functioning of this service since a complete description of the SHIPMAI tracking mechanism is detailed in [38].

Tracking an agent means following continuously all its movements in order to be aware of its current location at each time. Since mobility is a predominant characteristic of our agents, such service is important. Knowing the agent location is necessary for:

- Remote control of an agent: we cannot control it if we ignore where it is.

- Communicating with the agent: messages from the user or other agents could be forwarded to the agent via its hosting AEE.

- Agent sharing: if similar agent's work is to be done somewhere, it is important to know where this agent is located. So we can decide, based on performance criteria, whether it is better to assign the task to the same agent or send or clone another agent to do the work.

In SHIPMAI, tracking an agent is achieved by collaboration between the agent, the ADC and the AEEs. It is the controller that keeps the information about the agent location in persistent storage with other agent attributes.

We have seen in the communication section that an agent should ask for the authorization of the ADC before migrating. Thus the controller is already aware of the next location of the agent. However, the location variable is not definitely updated until the destination AEE receives the agent and informs the ADC.

3.5.4. Advantages & Drawbacks

The persistence strategy in SHIPMAI has two major advantages. The first one is the fault tolerance facility. That is, when partial or total alteration of agent data occurs, the work accumulated by the agent will not be lost. The system could easily recover by building an agent to replace the one tampered. The second positive point is the enhancement of system performance by performing parallel processing whenever possible: Agent clones can be sent in parallel to different hosts to perform independent tasks.

In existing platforms, assigning the persistence responsibility to AEEs leaves a lot to be desired concerning the robustness of the system. In fact, requiring all AEEs in the agent-space to provide a solid data management for agents seems to be far fetched. Investing heavily in implementing this management only in one server of the domain (ADC) is more logical and more economic.

Being a depository of valuable information, the ADC constitutes a vulnerable point in our system. It can be a target of choice for mal intentioned hackers. That is why this precious container should be carefully protected.

3.6. Security

3.6.1. Principles

The security issue is actually the greatest barrier still preventing the agent technology from widespread use and general acceptance. Promises hold by the mobile concept are not realizable unless efficient and practical security measures are taken.

In this perspective, security attracts very active research efforts in the agent community.

Traditional security mechanisms, found in static distributed systems, rely mostly on cryptographic techniques to implement authentication, authorization and access control. However, MA systems need, beside these classical measures, other elaborate provisions in order to protect the work and cargo of agents against hostile AEEs and also to protect hosting environments from malicious agents [39].

In our survey of existing mobile agent platforms, we have seen that most of them show gaps in the security issue and its implementation. Maybe this is because they generally consider it as an add-on to think of after finishing the design the system.

In our opinion, security reasoning should be part of the system design process since the early phases.

In fact, in SHIPMAI design, security concern was among the strongest reasons that brought the idea of partitioning the network into private agent domains. Following this philosophy, applying security techniques to the system components become less complex since the architecture is already adapted to handle them.

This section presents and analyzes how our infrastructure deals with this critical issue and how the architecture design helps coping with its challenge.

3.6.2. Analogy with Intranets

3.6.2.1. Privacy

In other platforms, agents and hosts do not have any prior knowledge about each other. This leads to one of the following situations:

- Agents and hosts can trust each other. In which case, it will be very easy for each one of them to cause damage to the other.

- Agents and hosts definitely do not trust each other. In which case, efficient security measures will be very difficult and expensive to implement.

In SHIPMAI, the vision is different. Agents and AEEs belonging to the same domain can have a certain degree of trust in each other, and thus will only require the use of elementary precautions in their interactions. A SHIPMAI domain becomes a private area where agents can be executed and agent data be transported with a minimal cost spent on security measures. The burden of applying elaborate security checking and controlling the access to the pseudo-trusted zone is left to the ADC as described in the following paragraph.

3.6.2.2. ADC as firewall

The ADC is for a domain what the firewall is for an organization Intranet. It is responsible for applying rigid security policies towards foreign entities wishing to access its domain of control. Policies include message filtering, agent authentication, mission verification, etc. The degree of control depends on the nature of the incoming message and its potential harm to the system. (e.g. a message containing an agent will undergo more security measures than a simple

request message). Furthermore, the ADC is in charge of protecting outgoing messages against tampering, especially agent messages transported outside the domain. It transfers them via secure channels using different security mechanisms.

The following drawing (figure 3.7) summarizes the security philosophy of SHIPMAI and the capital role played by the ADC in safeguarding its domain.

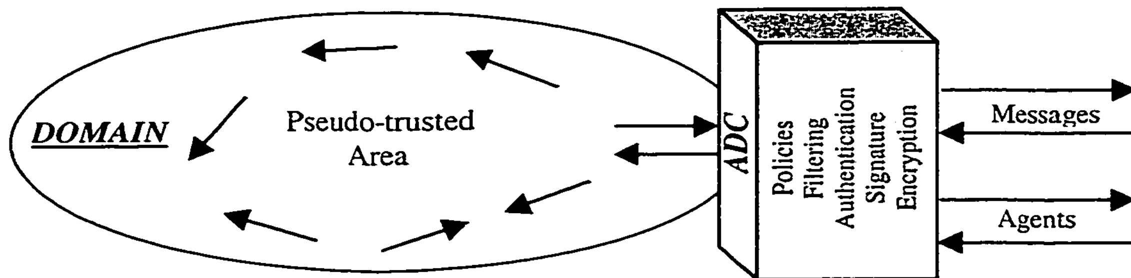


FIGURE 3.7: SHIPMAI SECURITY MODEL

3.6.3. Protecting agents

3.6.3.1. Potential harms

As discussed in [19], there are no satisfactory solutions to completely prevent a hostile AEE from causing damage to agents it hosts. In fact, being a simple piece of data that is inactive by itself, an agent constitutes a very easy target for a malicious environment which has full control of it. Such AEE can attack by:

- Not running the agent code correctly.
- Not providing enough resources to the agent.
- Preventing the agent from sending and receiving messages.
- Refusing to transfer the agent.

- Tampering with agent data [40].

3.6.3.2. Solutions

Recent elaborate research efforts aim to make possible the execution of encrypted code in order to prevent mischievous execution and make it difficult to use the agent for purposes other than those it was intended for.

In SHIPMAI, The ADC central persistence strategy provides an indirect solution for such security problems: By caching copies of agent code and regularly receiving its execution state and work results, the ADC can at each time reinitiate a tampered agent after a network failure or an AEE attack. The reinitiated agent will not lose all its accumulated work, but would restart from the last execution checkpoint reached before being attacked by the hostile environment.

Furthermore, the existence of ADCs as privileged authorities in their respective domains should reduce the probability of such AEE hostilities. That is, the ADC has the power to deny the legitimacy of an AEE and prevent it from hosting and executing agents. This is possible since the ADC controls all agents' movement inside its domain.

We can of course discuss the trustworthiness of ADCs themselves, however exaggerated it is. In this perspective, we engage further research work to extend the SHIPMAI hierarchy by adding a higher level. This level will consist of a few super-ADCs that supervise sets of domains, and whose credibility would be universally uncontestable.

3.6.4. Protecting hosts

3.6.4.1. Potential harms

Mobile agents are unknown programs that move from host to host to perform works. It is evidently dangerous for a host to execute a malicious agent and give it accesses to its resources.

Agents can be either engineered from scratch for malicious intent or captured and modified by active hostile entities.

People working in Mobile Agent Facility (MAF) identified the following types of active or passive attack from agent to host:

- Spamming: means overloading the host.
- Spoofing: means falsifying its identity to get access to critical resources.
- Trojan horse: means hiding or disguise its true intent.
- Replay: means copying legitimate operations.
- Eavesdropping: the passive attack of spying for the count of third parties.

3.6.4.2. Solutions

In SHIPMAI, the introduction of the hierarchy notion gives new perspectives regarding the security modeling. As aforementioned, access to a private area is strictly controlled by the ADC, which protects it against incoming foreign entities.

The identity and the mission of a foreign agent are verified by the controller and approved by home ADC of the agent. This restricts the probability of hostilities like spoofing or Trojan horse. Also, controlling outgoing messages helps preventing the possibility of eavesdropping.

Applying such measures at the ADC level has the advantage of lessening AEEs' onus and freeing agents to roam more easily inside the domain.

Preventing the other types of attack can be realized by a fine grained and extensible access control policy at the AEE level. The agent will only be given access to limited resources allowing it to perform its legal task.

3.6.5. Security mechanisms

The security mechanisms implemented in SHIPMAI will be described in detail in a dedicated section of the next chapter. For now, we will just enumerate them. First, there are security measures provided by the Java Virtual Machine (JVM) itself, which mainly concerns code execution control and resources access. Besides, we implemented some known techniques for data protection such as encryption for confidentiality, signatures for integrity, and authentication by digital certificates.

3.6.6. Advantages & Drawbacks

To summarize, we can say that the key force in the security model of our architecture resides primarily in its conception. We profit greatly from the analogy between SHIPMAI domains and Intranets. Such similitude allowed us to superpose and adapt easily the widely accepted Intranet security concepts to our MA platform.

By assigning the ADC the application of the major security measures such as authentication and encryption, we let AEEs focus more on executing agents and better controlling the access and the use of their resources. We also economize the time that an agent has to spend in a given domain, allowing it to achieve its mission with minimal cost.

Furthermore, the fact that agents should necessarily belong to a well-defined domain and be under the authority of a known ADC would make it difficult for vandals or impostors to send their hostile agents to roam domains and damage hosts. If such vandalism occurs, domain

controllers could deny the legitimacy of the guilty agents and users, and inform other controllers of their maliciousness.

As any centralized approach, the SHIPMAI security framework represents a minimum risk. That is, any failure in the ADC server or lacuna in its security implementation can have bad consequences on the domain welfare.

We can found our judgement of the security model in SHIPMAI on better grounds when we detail its technical implementation and practical deployment in the next chapter.

Chapter 4

4. SHIPMAI IMPLEMENTATION

4.1 Introduction

This chapter details our implementation of the SHIPMAI system. We start by explaining our choice of the programming tools. Then, we will describe the concrete constitution of the agent object. The next two sections deal respectively with the implementation of the messaging system and the security framework of SHIPMAI. We will also present some prototype screens.

4.2. Implementation choices

4.2.1. Why JAVA as programming language?

Choosing an appropriate programming language for the development of a mobile agent infrastructure is a very important matter. Such language should, at least, adequately support three aspects. These are portability, easiness of distributed programming, and efficiency of security mechanism. Java, at the time being, seems to be the only programmatic language to fulfill these requirements.

Concerning the portability, Java, thanks to its "byte code", could be supported by all the major computer platforms in the marketplace. In fact, any device that supports the Java Virtual Machine can interpret the Java byte-code. Java constitutes a cross platform environment destined for internationalization. The slogan "Write once, run anywhere" is often used to describe the

portability of Java. This portability is particularly interesting in the MA paradigm where agents are required to roam in the international network and operate in different environment. In an MA infrastructure such as SHIPMAI, the use of a portable language like Java, will allow us not to worry about platform heterogeneity, and focus more on agent-related problems and issues.

Java has also brought distributed software development to the world community in ways the computer industry has never seen before. Java class libraries come with support for network communications. They have several groups of classes to support the WWW and TCP/IP socket communications. These standard APIs are part of the core libraries ensuring that any platform with the JVM will be network ready. Hence, Java provides a great easiness in network programming. This is very important for the deployment and the popularization of mobile agents since all their applications are network-based. Beside this facility, Java supports multi-threading and dynamic class loading in its language specification and with its class libraries. This is very beneficial for AEEs that are appealed to host and execute several agents at the same time.

The third point is security. Agent hosts have to execute a foreign code coming from unknown sources. Thus, strict security measures should be taken to protect these host servers. In Java, security checking is built into the JVM in order to enforce memory safety and protect local resources. In fact, the JVM enforces the type safety of the language, preventing programs from accessing memory or calling methods without authorization [41]. Existing JVM implementations also enforce a simple "sandbox" security model. The sandbox model isolates mistrusted code in a restricted environment in order to control its execution and to prohibit it from using any sensitive system services.

Because it satisfies these three requirements, Java has been chosen as the programming language for SHIPMAI. We use the standard JDK1.2 (Java Development Kit).

Nonetheless, Java has some other positive points that make it appropriate for the MA paradigm. For example, except for a few primitive data types, Java is a pure object oriented language. All data is represented as objects, which is in favor of the notion of abstraction and autonomous behavior claimed by agent technology partisans. Also Java is widely accepted through the world, and has a great support from the computer and telecommunications industries. The MA paradigm can profit from this wide acceptance to gain more popularity.

4.2.2. Why Directories & LDAP for data management?

In SHIPMAI, there is a need for a storage system in order to store users' information, agents' attributes, and AEEs' data. This system should be simple and easy to use for the manipulation and the update of data. Also, because of the distributed nature of MA-based applications, such system should also support a data distributed scheme across several machines as well as an easy data access through the network. These are the main reasons that drove us to choose directories as means for data storage, retrieval and update.

A directory is like a database, with a tendency to contain descriptive and attribute-based information. It supports the storage of all kind of data. Directories do not usually implement the complicated transaction or roll-back schemes used in regular databases. They are also tuned to give quick-response to high-volume lookup or search operations.

These above points are particularly interesting for us. First because most the users and agents' information we need to store can be presented under the form of "***attribute : value***". Second, the update simplicity helps in not affecting the servers' performance, especially at the ADC level. Finally, quick response to lookup requests is required because of the high solicitation of agents' information and results by users and applications.

As a protocol to access the data, we use the Lightweight Directory Access Protocol [42]. LDAP, also described in RFC 1777 [43], is a network protocol for accessing directory information. The LDAP standard defines also an information model defining the form of the data to store as well as a namespace defining how information should be referenced and organized. This makes LDAP both an information model and a protocol for querying and manipulating it. Another main advantage of the LDAP protocol is that it is designed to run directly over the TCP/IP stack, which makes it easily usable with Internet-based applications like MA-based ones.

The LDAP directory service model is based on entries. An entry is a collection of attributes that has a name, called a distinguished name (DN). LDAP provides a set of standard attribute types based on the X500 notation [44], but one may also define his own syntax, attributes, and even object classes. This allows one to tailor his directory depending on its specific requirements and needs. In LDAP, entries are arranged in a hierarchical tree-like structure that reflects political, geographic and/or organizational boundaries. Figure 4.1 illustrates this structure:

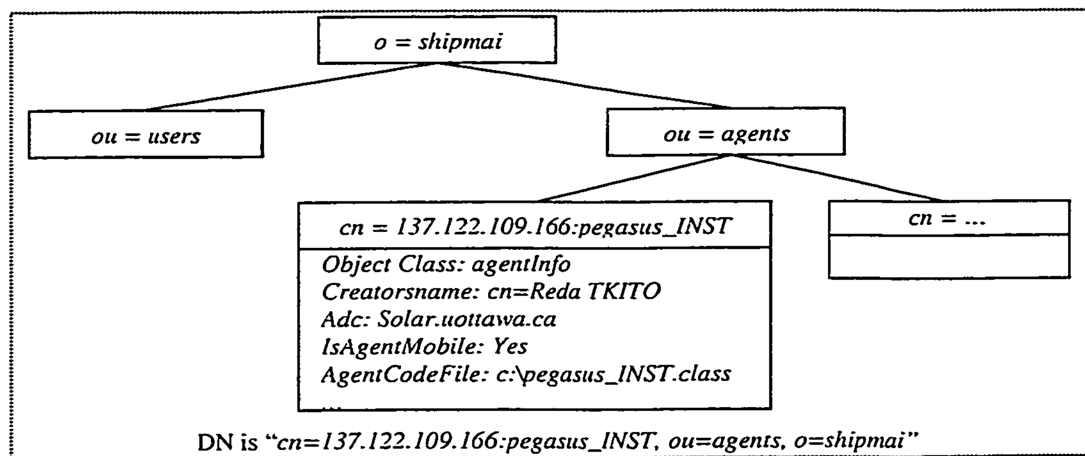


FIGURE 4.1: LDAP DIRECTORY TREE EXAMPLE

As a product, we are using Netscape Directory Service [45] because it provides Java APIs for manipulating data and uses the popular Netscape browser as an LDAP client.

4.3. Agent object

4.3.1. Components

In SHIPMAI, an agent is a collection of Java objects. In our implementation, we have a main class called ***SHIPMAI_Agent***. It contains the methods allowing the agent to communicate with platform *(platform_code)*. That is, to ask for migration, to send results to the ADC, etc. This agent class has an aggregation relationship with two other Java classes instantiating the *Identity* and *Mission* objects. In what follows, we will describe only the agent attributes. The methods could be found in the class diagram.

4.3.1.1. Identity

It is common also to AEEs and ADCs and it consists of the following attributes:

- ***Type***: since all SHIPMAI elements have an *Identity* (see messaging implementation), this "*type*" field specifies if the entity is an ADC, an AEE or an agent (which is the case here).

- ***ADC***: it is the controller of the home domain. Here, it is defined by the server name or IP address, and the port number on which it is running.

- ***Local_Name***: a string chosen by the agent owner. It represents an identifier that should be unique inside a domain. If it is already used, the user is asked to choose another name.

- ***Global_Name***: it is a name that should uniquely identify the agent in the whole network. In SHIPMAI, it is automatically generated by a combination of the two-aforementionned fields [38]. e.g. if the IP address of the *ADC* is *137.122.109.166*, and the agent *Local_Name* is *reda_agent*, then the global_name will be *137.122.109.166:reda_agent*.

- ***Certificate***: an X509 certificate chain generated by the ADC upon agent registration.

- ***Mobility***: a boolean stating whether the agent is mobile or not.

- ***Location***: it is the AEE machine representing the fixed location of the agent if it is static, and the location to which it should normally return once its mission accomplished if it is mobile.

4.3.1.2. Mission

It describes the application-related mission of agents. Its most important elements are:

- ***Itinerary***: the set of hosts the agent should visit. It is implemented as a stack from which the agent pops its future destination.

- ***Jarfile***: the application source code of the agent represented as a JAR file. It is loaded at the AEE by the SHIPMAI *classloader* before being run by the JVM.

- ***State***: it describes the agent state (i.e. active, suspended,...). Also, it specifies the checkpoint the agent has reached in its execution.

- ***Results***: since it is dependent of the application, this field consists only on a vector of objects where the agent can store its accumulated work under different formats.

4.3.2. Class diagram

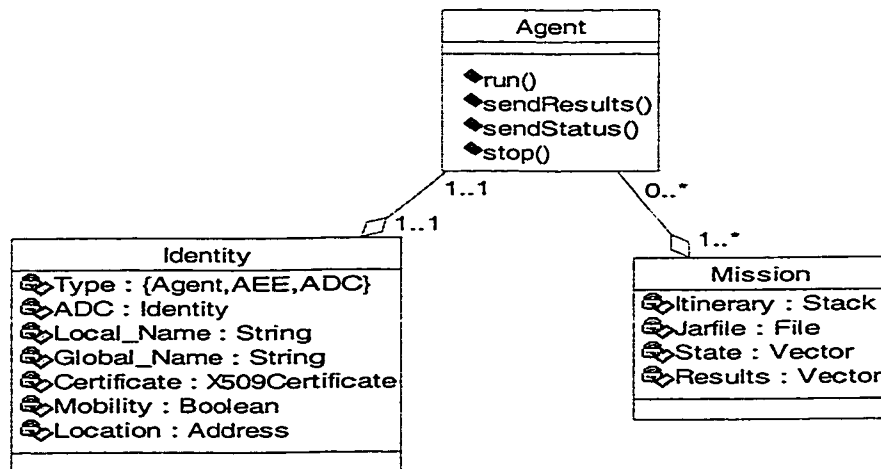


FIGURE 4.2: UML VIEW OF AGENT CLASS DIAGRAM

4.4. Messaging system

4.4.1. Components

SHIPMAI provides an efficient mechanism for different entities (i.e. ADCs, AEEs and agents) to communicate with each other using a uniform messaging system. This mechanism is implemented in Java on the basis of four building blocks: *Identity*, *Message*, *Message handler* and *Registrator*.

4.4.1.1. Identity

It is an instantiation of the same class used in the agent composition described in the previous section. The object *Identity* identifies all sender and receiver entities (i.e. ADCs, AEEs and agents) by giving their name, location (i.e. IP address and port number) and their certificate chain. Of course the location is valid only for static components and agents. Thus mobile agents must rely on the host environment (AEE) for receiving their messages. The certificate chain, which is used mainly for security purposes, gives the entity's public key and the certificate chain of the signer.

4.4.1.2. Message

The *Message* object is the heart component of our messaging system. It mainly encapsulates the following elements:

- The identity of the message sender.
- The identity(ies) of the message recipient(s).
- The name or type of the message, which should be significative. For example, a message for agent migration would be called *"migration"*.

- A vector of objects called *content* in which all the contents of the message will be put. Would it be a specific request, work results or even agent objects. According to the name of the message, we will have a method for treating this message content.

The *Message* class provides an API for constructing and customizing messages. Also the routing mechanism is built inside the *Message* class: the API includes a ***send()*** method that delivers the message to its destination(s) according the infrastructure hierarchy.

4.4.1.3. Message handler

Message objects are sent to the recipient's appropriate queues or *Message_handlers*. Of course, only static entities have *Message_handler* objects.

The *Message_handler* has a server socket opened in a given port number, and is continuously listening for incoming messages. At the reception of a message with a given name, the handler dynamically invokes the method, with the same name, on the registered object (See paragraph 4.4.2.1.).

In case where the message is destined to an entity outside the domain, it transits via the ADC, whose *Message_handler* calls a method that will forward the message to the destination domain's ADC after performing security measures.

4.4.1.4. Registrator

In SHIPMAI, entities should subscribe for messages they are interested in. The *Registrator* class is responsible for registering objects for message types they would be asked to treat. The registration to a certain massage can be done at system initialization or during functioning.

Every *Message_handler* has a correspondence table managed by a *Registrator* object. This table (implemented as a Java Hashtable) associates each message name with the Java object containing the method that would take care of the message treatment.

The following figure (figure 4.3) summarizes the message passing mechanism implemented in SHIPMAI and the way the message is treated.

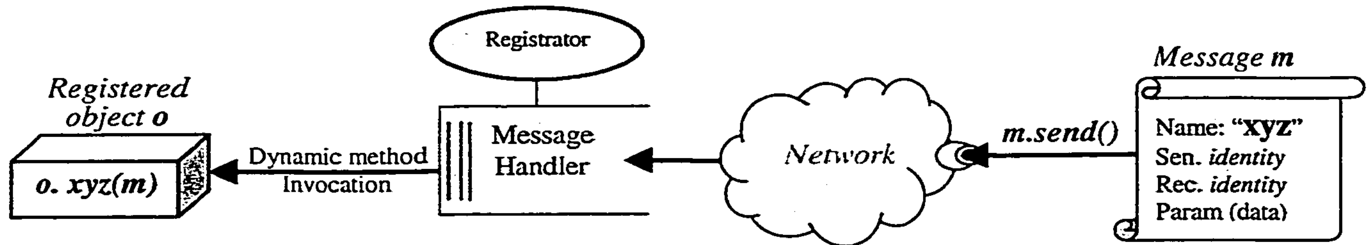


FIGURE 4.3: SHIPMAI MESSAGE TRANSMISSION AND HANDLING

4.4.2. Techniques & Tools

4.4.2.1. Java Reflection for dynamic method invocation

In SHIPMAI, we correspond to each type of message a specific method with the same message name, that would take care of the treatment of the message and performs the adequate processing.

SHIPMAI uses a dynamic invocation of methods thanks to the Java reflection API [46]. This API is destined to sophisticated applications that need to discover at will the members declared by a given class. Such applications need runtime access to the implementation of a class at the level provided by a class file.

We use mainly the two Java classes: ***Class*** and ***Method***. The scenario is the following:

When the message Handler receives a certain message with a given name, it checks its registration hashtable to know which object is registered to such message name. After that, it

invokes the Object ***getClass()*** method to retrieve the class of this object. Then it uses the ***getDeclaredMethod()*** method of the ***Class*** Java class to get the ***Method*** object desired. The argument passed to ***getDeclaredMethod()*** is the name of the message (because we give the method associated to a given message the same name as the message itself). Once it gets the Method object, the message handler can invoke dynamically the appropriate method by just calling the ***invoke()*** method on the ***Method*** object and giving it the ***Message*** object as parameter.

The following code fragment summarizes this processing:

```
Message receivedMessage = myMessageHandler.receive();
String messageName = receivedMessage.getName();
Object registeredObject = myMessageHandler.getRegisteredObject(messageName);
Class objectClass = registeredObject.getClass();
Method appropMethod = objectClass.getDeclaredMethod(messageName,Class[]{Message});
AppropriateMethod.invoke(registeredObject, Object[]{receivedMessage});
```

The main advantage of using this dynamic method invocation is that Message Handlers does not have to know anything neither about the message nor about the method that should take care of it. Thus, we produce a decoupling between the reception and the treatment of the message.

4.4.2.2. Comparison with RMI

In SHIPMAI, all communications, agent transfers, and service requests are based on the transmission of ***Message*** objects. Thus, accessing any service can be done just by sending the appropriate message to the receiver queue or message handler. The latter would invoke the concerned method. Comparing this messaging mechanism with RMI (Remote Method Invocation), we find two main advantages for our implementation.

- First, the SHIPMAI dynamic method invocation explained before allows easily the adding of new message types, and thus new services, without touching the code of the Message Handler. It suffices to program the appropriate method in the concerned object and register it. In

RMI, adding a new service would necessarily imply an ***rmic*** processing (Java RMI Compiler) to generate stubs and skeletons for objects interested in the new service. Furthermore, service requesters should carry all the stub files associated with the services they are registered for. This is particularly compromising for mobile agents that have to be the most lightweight possible due to their mobility behavior.

- Second, the method invocation in RMI is of synchronous nature, which means that a connection problem would lead to an invocation exception. Avoiding the complete failure of the operation would imply building complex caching mechanism on top of RMI. Mobile agents are by definition destined to operate efficiently in unreliable networks such as the Internet, and to tolerate connection intermittence. That is why asynchronous communication is more appropriate in an MA infrastructure. Our messaging implementation allows such asynchronous operating since the message object can be easily cached and resent in case of connection problem. Once the handler receives the message, the invocation is done locally.

4.4.3. Class diagram

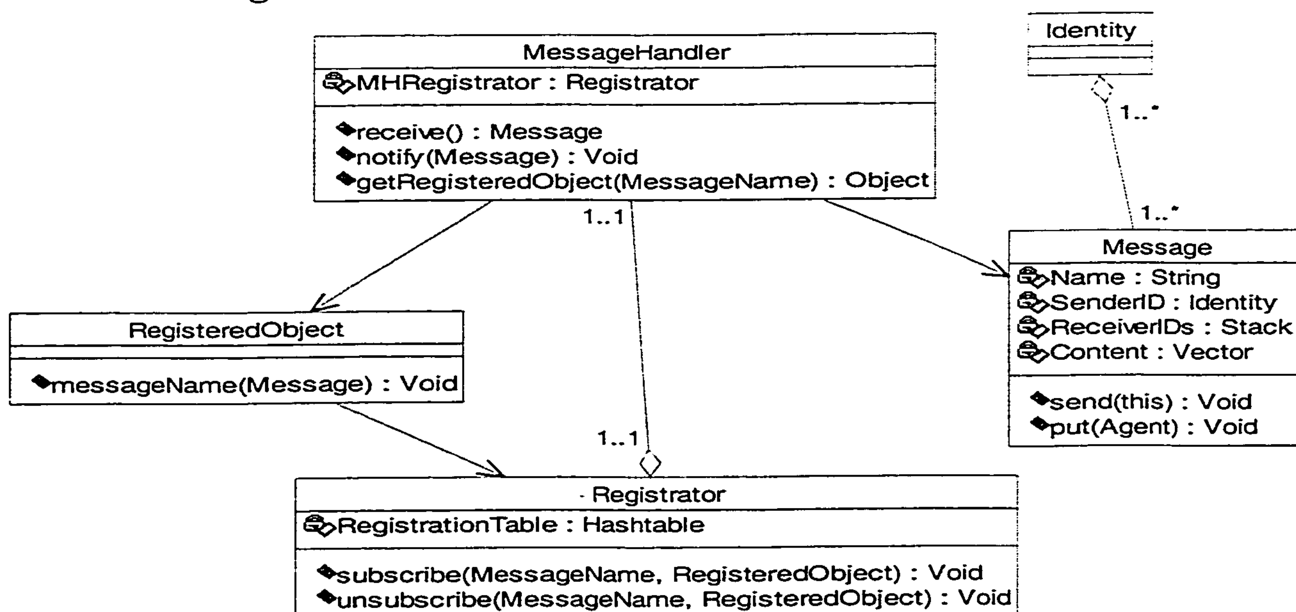


FIGURE 4.4: UML VIEW OF THE COMMUNICATION FRAMEWORK CLASS DIAGRAM

4.5. Security framework

4.5.1. Components

The ADCs act as firewalls or proxies for their domains. To secure data and agents circulating through them and to identify senders of the message, SHIPMAI makes use of different security techniques. These are also required for AEEs but with less degree of necessity since we consider a domain as a pseudo-trusted area.

In the implementation of our security framework, we distinguish between four components or building blocks. They are: Encryptor, Signer, Authenticator and Certificate_Issuer.

In this section, we will describe the role and functioning of each building block. In the upcoming section (4.5.2), we will detail the techniques and the tools that we used to implement these components.

4.5.1.1. Encryptor

The role of an ***Encryptor*** object is to protect the confidentiality of information. Encryption is the basis of a secure transmission over an open medium such as the Internet. It makes a message not understandable to anyone that does not have the key to decrypt it. In our system, the ***Encryptor*** intervenes in many situations such as during a message exchange between two ADCs when negotiating an agent transfer, or when an agent outside its native domain wants to send the results of its confidential work to its home ADC. Note that not all messages need to be encrypted. For agent code in some situations, it is more important to detect tampering than to protect confidentiality.

SHIPMAI uses asymmetric key encryption. The algorithm we use is the ***RSA*** public key algorithm (see "Techniques and tools" subsection).

4.5.1.2. Signer

The role of digital signatures is to prove the integrity of exchanged data. A transmitted message cannot be protected against alteration, but alteration could be detected if the sender signs the message. SHIPMAI agents, and particularly their codes, are signed before inter-domain migration, so that ADCs can detect their eventual tampering. In such case, they are not allowed to enter to the domain. Other critical SHIPMAI messages containing negotiation information for example should also be signed. A SHIPMAI ***Signer*** object is responsible for signing outgoing messages and verifying the signature of incoming messages.

A signature is an encrypted message digest that summarizes uniquely the message content. It is calculated by the sender ***Signer*** object and attached to the message. At the receiver side, it is verified by another ***Signer*** object. If the verification is positive, it means that the message received is identical to the one originally sent; otherwise, its content has been changed. In SHIPMAI, we use an algorithm called ***DSAwithSHA***. See "Techniques and tools" subsection.

Thanks to a Signer object, an ADC can be sure that an agent wishing to access the domain has not been corrupted by third parties during network travel. However, how could an ADC know the real source of an agent or a message? That is why sender authentication is to be done before verifying the signature and the integrity of a message.

4.5.1.3. Authenticator

How could one know that a given message came really from the source it claimed to come from and not from an impostor? Here, a SHIPMAI ***Authenticator*** comes into play. The functioning of the Authenticator is based on the use of digital certificates. A certificate is a sort of inforgeable identity card attesting that the public key shown in it belongs really to the identity

it carries. The Authenticator checks the identity of the sender from the certificate attached to the incoming message, then it extracts the public key and passes it to the ***Signer***. If the latter can decrypt the signature by the public key, this proves that it was encrypted by the sender private key. This normally excludes the possibility of impostors since the private key is to be kept secret by its holder.

In our infrastructure, an extensive use of certificates is made. Not only static entities have certificates, but also mobile agents. ***Authenticator*** objects verify the identities of message senders, and also those of incoming agents by checking their certificates issued by a ***Certificate_ Issuer*** object belonging to their home ADCs.

4.5.1.4. Certificate_ Issuer

A digital certificate is also a signed piece of data. It is signed by its issuer to guarantee its authenticity. The issuer of a certificate should be an entity whose legitimacy is incontestable and whose public key is universally known.

As SHIPMAI trusted entities, ADCs act as certification authorities (CA) for their domains. Hence, the ADC issues and self-signs its own certificate. It also issues and signs certificates for agents created by domain users and for AEEs operating under its control. These operations are performed by an ADC component called ***Certificate_ Issuer***. The kind of certificates used in SHIPMAI is the standard ***X509 certificates***.

4.5.2. Techniques & Tools

4.5.2.1. IAIK-JCE package

It is a product of the IAIK-Java Group, which is part *the institute for applied information processing and communications (IAIK)*. The group is mainly working on Java-Cryptography and

Java-Security. The IAIK-JCE (Java Cryptography Extension) package is a set of easy-to-use APIs and implementations of cryptographic functions of different types. It supplements the security functionality of the default Java JDK 1.1.x / JDK 1.2, including digital signatures and message digests with different algorithms. The package supports standard X.509 certificates.

SHIPMAI makes use of IAIK-JCE v2.5 to implement asymmetric encryption and digital signatures (with RSA and SHA algorithms). X509 v3 certificates are used for authentication.

4.5.2.2. RSA as encryption algorithm:

SHIPMAI uses the ***RSA*** *(Rivest, Shamir, Adleman)* public key encryption algorithm with 1024 bit-keys [47]. A key serves to parameterize for the encryption algorithm; i.e. one message ciphered by the same algorithm but with two different keys will produce two different encrypted pieces of data.

Symmetric key algorithms (using the same key for encrypting and decrypting), despite their fastness, have two major drawbacks. The first is that the key itself should be sent through a secure medium so that third parties cannot discover it and use it to decrypt confidential messages. The second inconvenient is that different keys should be used for different communication sessions; this becomes rapidly very hard to manage.

With asymmetric key algorithms such as RSA (using public and private keys), it is very easy to manage keys. The holder, say an ADC, has just to keep its private key secret and send the public one openly in the network. If an entity wants to send confidential information to the ADC, it encrypts the message with the ADC public key. The message can then be decrypted only by the ADC since it is the only one to have the private key. This will allow also the ADC to use the same key pair with different corresponding entities.

For these reasons, the SHIPMAI ***Encryptor*** uses asymmetric encryption. Particularly, we opted for the use of the ***RSA*** algorithm because of its universal reputation. RSA supports also key pair generation. It is described in PKCS#1 (Public-Key Cryptography Standards).

4.5.2.3. DSAwithSHA as signature algorithm:

This algorithm is a combination between ***DSA*** (*Digital Signature Algorithm*) [48] and ***SHA*** (*Secure Hash Algorithm*) [49].

SHA, defined in *FIPS PUB 180-1* (Federal Information Processing Standards Publications), serves to compute a digest for the message. The force of such algorithm is that for two pieces of data with only one bit of difference, it will produce two completely different digests. This allows easily the detection of any slight tamper of data. **SHA** produces digests of 160-bit size.

However, computing a message digest is not sufficient. In fact, after modifying the message, a vandal can replace the original digest by a new valid computed digest, which will make the tamper undetectable. Such situations are very dangerous in SHIPMAI since agents carry executable code that can be very harmful for hosts if tampered by third parties.

To avoid this, the message digest should be encrypted with the private key of the sender. In SHIPMAI, this is done by the DSA algorithm (described *in FIPS PUB 186).* In this way, even if vandals modify the message and compute a new digest, they will not be able encrypt it because they do not know the private key of the sender. Encrypting the digest with another key will make the tamper detectable because the sender public key could not decrypt it.

DSAwithSHA produces ciphered message digests or digital signatures. It is supported by IAIK-JCE and is directly used by the SHIPMAI ***Signer*** for issuing and verifying signatures.

4.5.2.4. X509 Certificates

Digital certificates are used in SHIPMAI for the authentication of agents, AEEs and ADCs. As mentioned before, Certificates are documents attesting to the binding of a public key to some entity. They help prevent the use of a phony key to impersonate another individual or entity.

In their simplest form, certificates contain a public key and a name. But because of their wide use, certificates also contains an expiration date, a serial number, the name of the certifying authority (CA) that issued the certificate and, most importantly, the digital signature of the certificate issuer. The most widely accepted format for certificates is defined by the ITU-T X.509 international standard [50]. SHIPMAI uses the last version of X509 certificates (v3). A X509 certificate is described using the ASN1 (Abstract Syntax Notation 1) as follows:

```
Certificate ::= SEQUENCE {
    tbsCertificate          TBSCertificate,
    signatureAlgorithm      AlgorithmIdentifier,
    signature               Bit string }
```

In this sequence, *signature* is the digital signature of the certificate performed by the CA (which is an ADC in SHIPMAI). The *signatureAlgorithm* is the identifier of the algorithm used for signing the certificate. TBSCertificate is the "To Be Signed" X509 certificate that is also described as an ASN1 sequence structure.

```
TBSCertificate ::= SEQUENCE {
    version                 [0] EXPLICIT Version DEFAULT v1,
    serialNumber            CertificateSerialNumber,
    signatureAlgorithm      AlgorithmIdentifier,
    issuer                  Name,
    validity                Validity,
    subject                 Name,
    subjectPublicKeyInfo    SubjectPublicKeyInfo,
    issuerUniqueID          [1] IMPLICIT UniqueIdentifier OPTIONAL,
                            -- If present, version must be v2 or v3
```

```
subjectUniqueID     [2] IMPLICIT UniqueIdentifier OPTIONAL,
                    -- If present, version must be v2 or v3
extensions          [3] EXPLICIT Extensions OPTIONAL
                    -- If present, version must be v3    }
```

In SHIPMAI, the ADCs are the certification authorities for their domain entities (AEEs and native agents). An ADC public key is known by other Controllers, so that verification of AEEs or agents' certificate signatures can be easily performed. Since certificates have a limited validity period, the ADC, as a CA, should maintain a CRL (Certificates Revocation List) for the certificates that had expired or whose validity are denied for other reasons.

The following is an example of an AEE's X509 certificate stored in a ***.der*** file.

```
Version: 1
Serial number: 3
Signature algorithm: dsaWithSHA
Issuer: C: CA , O: MMARL , OU: SHIPMAI , CN: ADC:SOLAR (CA)
Valid not before: Thu Jun 03 03:31:29 EDT 1999
      not after: Fri Dec 03 03:31:29 EST 1999
Subject: C: CA , O: MMARL , OU: SHIPMAI , CN: AES:MEDIAPRO2
Sun DSA Public Key
Parameters:DSA
p: 827dd49c a2056984 e98371b1 340d5d71 839285b2 5acaa382 d7ac386e 9440843f
   0a467aa8 75a8c1ca 3b70ba6a 970712f6 b199ed3e ec5313f3 940a67bb d69f3872
   2961ab02 3d17a133 3c52235d 9fb7d10e 95e3a55e f9b04fc7 c920c572 da7ac3d5
   0f240dbb 8e54da9e bb702111 c53582e5 35852e9f 593979b3 3250c886 83961917

q: fa5079da fa3f3ab1 e80a6df5 bd16f224 d8f8d71b

g: 4fbdf52e 3304f051 c17ca55c 9381b5c1 7d4c2050 76853450 cfd9fc72 b2e1b2b1
   6fa01048 b8ff17e7 a90ae1e0 18053e34 d9d561df 714cc8dc 92b151b5 df665970
   6b5e57c3 19a2d658 3b7d32d2 e9e1f166 3eaaac46 0dcd4e67 7036f7f9 be0b2e16
   a05d695d 5b8113a9 03cb3863 561abd36 4a5d6c15 6617fa10 a32099e1 d2347713

y: 3404a4ca 508b19ec 8b68007a efe0eaf6 bbbf853e 5dccb3e4 0e1eef3a 4e0262d2
   1581c741 89cbdfdb 4b3a9cbd 57fef470 9030d55f 95252fe5 ae015740 8300bff8
   7ef294f1 beb35baa 90c7012e 22e79753 259486ef 030cce61 a940fc1b dd983ac3
   77655f16 7505dcff e9035abd 25e61a49 2d092cee 65f2d179 25fd547c 92787668
Certificate Fingerprint: 33:06:0B:42:D3:D2:6B:59:A2:9A:4D:70:B3:89:19:F3
```

4.5.2.5. Secure Socket Layer

In SHIPMAI, we chose to use asymmetric encryption although its processing take longer than symmetric encryption processing. This is because most communications in SHIPMAI are asynchronous and short. However, in the rare cases of real time and long message exchange, such as when two ADCs negotiate an agent transfer, SHIPMAI makes use of the Netscape SSL technology [51]. SSL is a generic way to secure long peer communications. It combines all security techniques in an optimized way, using encryption, signatures and certificates. After mutual authentication by certificates (this is called SSL handshake), one of the two ADCs, let us say ADC1, chooses a symmetric key and sends it to ADC2 using asymmetric encryption with ADC2's public key. For the rest of the connection, fast symmetric encryption is used during the exchange of signed messages.

4.5.3. Class diagram

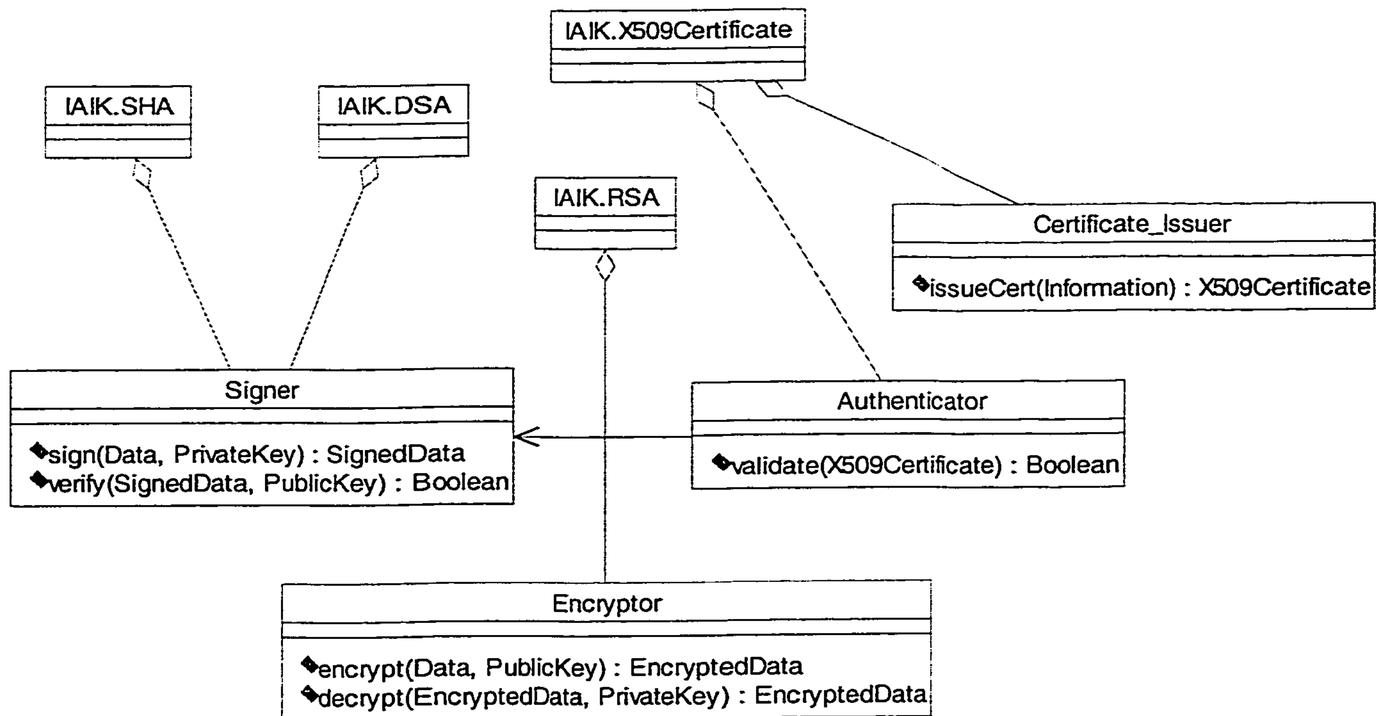


FIGURE 4.5: UML VIEW OF THE SECURITY FRAMEWORK CLASS DIAGRAM

4.6. Prototype

4.6.1. Agent creation

A registered user can create its own agent via an applet implemented at the level of the User Application Interface (UAI). The following GUI screen illustrates this operation (figure 4.6).

FIGURE 4.6: GUI SCREEN FOR AGENT CREATION REQUEST

4.6.2. Agent directory storage

Once the agent is created upon a user's request, it is registered in the ADC. Then its fundamental components are stored in an LDAP directory server as an entry with the organizational unit ***"ou=agents"*** and the organization ***"o=shipmai"***, which is the root of our SHIPMAI LDAP directory tree. The following figure (figure 4.7) shows the agent directory viewed by the Netscape browser, which plays the role of an LDAP client.

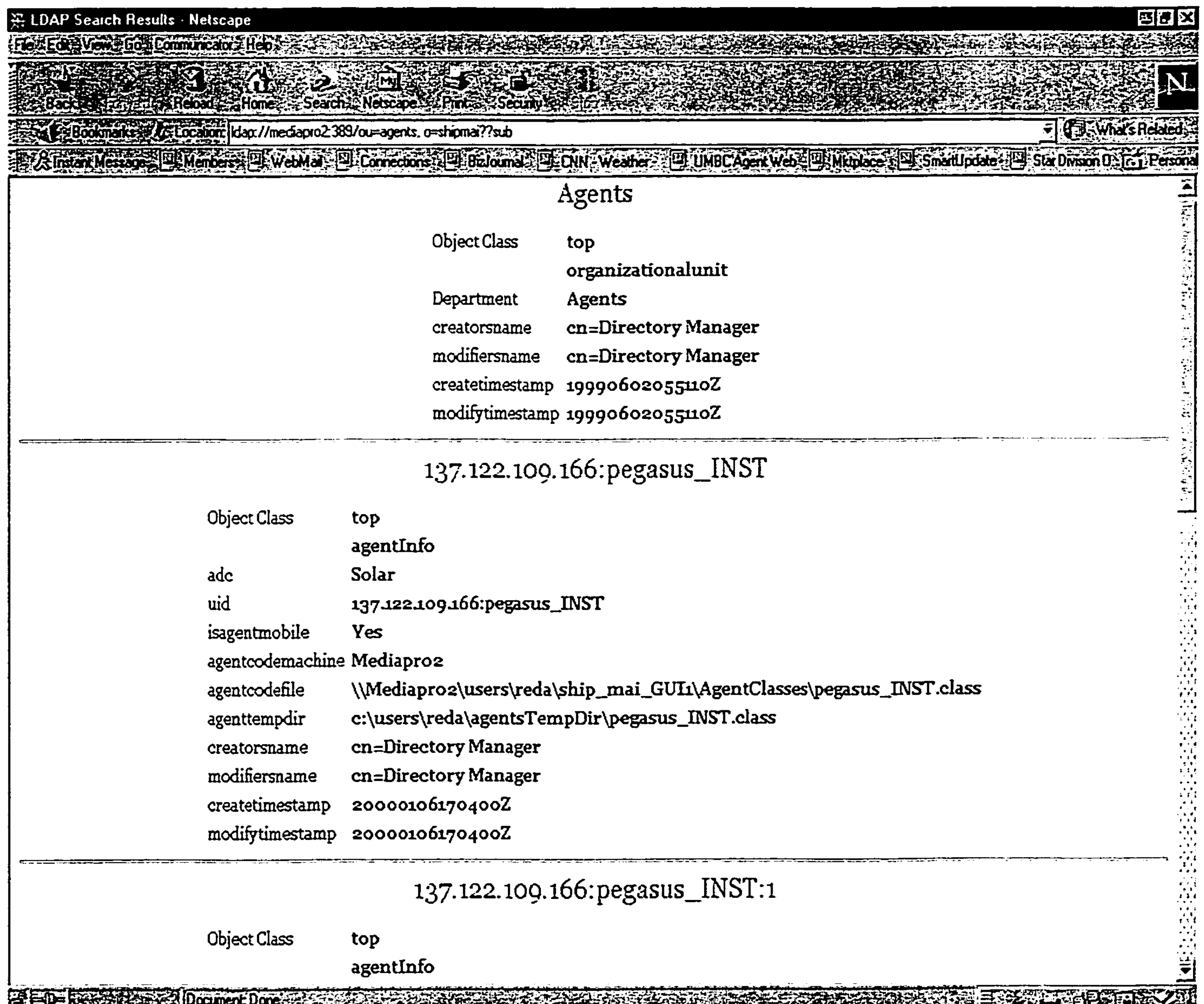


FIGURE 4.7: GUI SCREEN FOR AGENT DIRECTORY

4.6.3. Intra-domain & Inter-domain agent migration

The following two sequence diagrams (figures 4.8 and 4.9) illustrate by method calls the intra-domain and inter-domain migration processes that we explained in the precedent chapter.

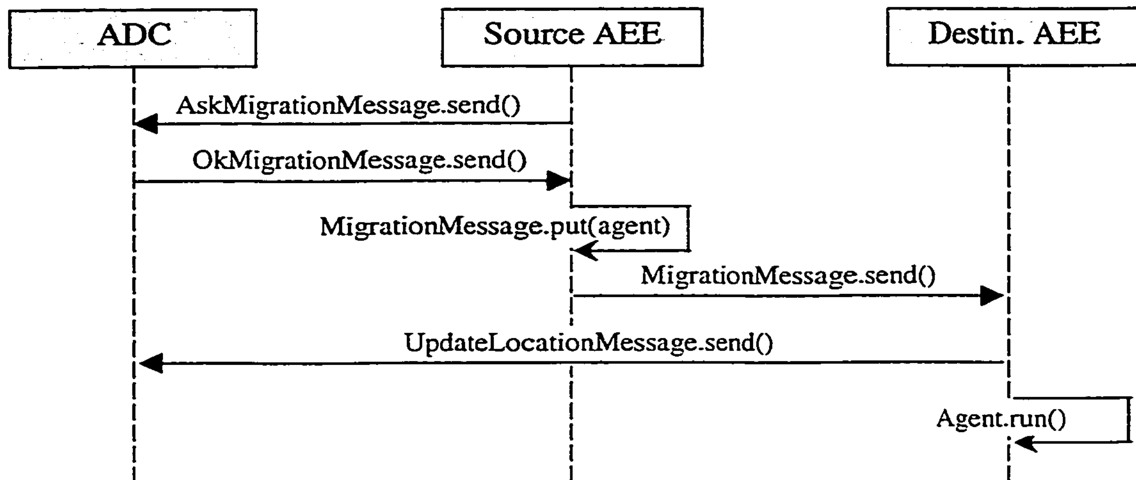


FIGURE 4.8: UML SEQUENCE DIAGRAM FOR INTRA-DOMAIN AGENT MIGRATION

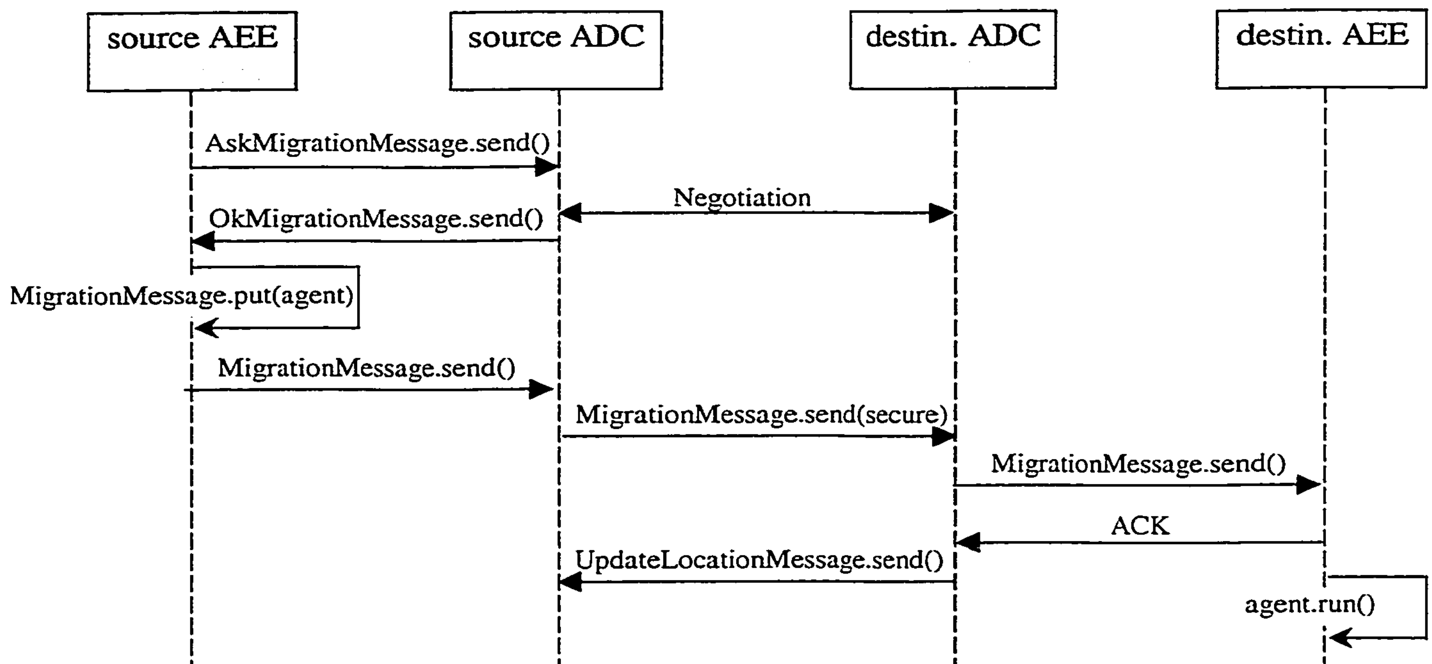


FIGURE 4.9: UML SEQUENCE DIAGRAM FOR INTER-DOMAIN AGENT MIGRATION

Chapter 5

5. MABSD APPLICATION

5.1 Introduction

In this chapter, we will first explain our motivations behind the development of a software distribution application. The OSD standard used for MABSD will be briefly introduced before going through the description of the application architecture and its components. We will end the chapter by presenting the implemented prototype and showing some of its GUI screens.

5.2. Motivation

Nowadays, with the evolution of distributed computing and the increasing heterogeneity in resources, the software distribution and acquisition have become a challenge for enterprises and individuals. In fact, this complex distributed environment leads sometimes to costs of manually delivering, installing, and administering software even greater than the purchase price of the software itself. Besides, manual distribution via floppies and CDs completely obey the "pull" paradigm, where every action is user initiated.

On the other hand, the massif migration of computers to a networked environment makes the network (especially the Internet) a powerful medium for software distribution. Such automatic strategy can substantially reduce the total cost as opposed to the manual method. Moreover,

network distribution may use a "push" paradigm, which provides a hands-free install and an automatic version upgrade.

Nonetheless, using Internet delivery does not resolve all software distribution issues. There are still problems of failing downloads due to disconnection or freezing. Statistics show that approximately half of Internet downloads are never installed, which frustrates users and thus affects the distributor's revenues.

5.3. Objective

Having in mind the resolution of the above problems, we have designed and implemented automatic software distribution application that we baptized MABSD (Mobile Agent Based Software Distribution).

Our belief is that the mobile agent paradigm could be very well suited for the conception of such type of applications that are network-based. In fact, a smart use of the agent technology should allow an efficient and reliable package distribution and could eliminate completely the failure problems. MABSD aims to automatically deliver, install, and update software on client machines scattered in a heterogeneous network with a minimum user intervention.

Another objective of the MABSD is to serve as a validating application for the SHIPMAI platform. The purpose is to test our infrastructure's ability to support useful applications. We think that SHIPMAI provides a good supporting infrastructure for this application because of its advanced features in term of security and fault tolerance. Nonetheless, the overall design of MABSD is of a high level of abstraction and could easily be adapted to suit any other agent platform.

5.4. Open Software Description (OSD)

5.4.1. Definition

OSD [52] is a language or a vocabulary for describing software packages, their composition, their attributes and their dependencies in a standard and platform-independent way. The term package is an open-ended term that can be used to refer to applications, plug-ins, applets, *JavaBean* components, etc. OSD only describes software packages ; their code can be written in any programming language.

The first key force of OSD is the fact that it is an application of the eXtensible Markup Language (XML) [53], which witnesses an exponentially increasing popularity these last years. The OSD format, which is a W3C standard proposed by Microsoft and Marimba, is expected to be useful in automated software distribution environments.

5.4.2. Representation

OSD is defined using XML, the emerging standard for representing Internet data. XML provides a general method of representing structured data in the form of lexical trees. Since OSD is an XML-based vocabulary, it uses the same data model. OSD files are expressed in plain text where package's descriptive data is enveloped with markup tags represented as tree elements. The relationships between tags can be expressed as combination of the three elementary relationships: *parent-of*, *child-of*, and *sibling-of*.

OSD provides information about different software characteristics, ranging from the package title to the required operating system and the interface language. In addition, It specifies software dependencies as a directed graph where nodes are packages' OSD files and arcs are directed cross references determining which components requires the installation of the other.

5.4.3. Specification

The OSD specification consists of two broad categories of tags: major elements and minor elements. A minor element is a child or a descendant of a major element.

The three OSD major elements are *SOFTPKG*, *IMPLEMENTATION* and *DEPENDENCY*.

SOFTPKG is the root of the OSD lexical tree; and thus is *parent-of* the two other major tag elements: *IMPLEMENTATION* and *DEPENDENCY*. It provides a general description of the package with some attributes like NAME, VERSION and HREF that optionally indicates a web page associated with the software package distribution. SOFTPKG is *parent-of* other minor elements such as *ABSTRACT* and *TITLE*.

IMPLEMENTATION is used to describe an implementation of the software package. This description is done via some minor elements that are *child-of IMPLEMENTATION*. Among these children tags are *IMPLTYPE*, *LANGUAGE*, *DISKSIZE*, *PROCESSOR* and *OS* for the required operating system(s).

DEPENDENCY serves to indicate a package dependency necessitating the installation of another software component. As a matter of fact, *DEPENDENCY* can be *parent-of* a *SOFTPKG* element containing the description of the required package. It is also parent-of a *CODEBASE* minor element that may be used to point to the URL for downloading the software archive.

5.4.4. Using OSD in MABSD

We use OSD in our software distribution application for many reasons:

- OSD is a W3C standard, and relates to other Internet standards like XML and CDF [54]. This somehow guarantees its extensibility and evolution with experience.

- OSD is simple and intuitive. Its design eases implementation and encourages adoption.

- Since MABSD uses mobile agents, we could take an important advantage from the software dependency description offered by OSD. Agents can achieve a cooperative work by getting and installing all the needed components with minimum user intervention.

- Because OSD is based on the XML vocabulary, the software description could be easily customized depending on the application needs. In MABSD, we have added our own tags, which does not belong to the OSD standard. These tags allow us to provide more information about a software package. Two of them are PRICE and WARRANTY.

The following (figure 5.1) is an imaginary example of an OSD file, shown for illustration purposes only.

```
<SOFTPKG NAME= "SHIPMAI"  VERSION= "1.0" >

        <TITLE> SHIPMAI agent platform </TITLE>
        <LICENSE HREF= "Copyright (c) 1998-2000, MMARL, all rights reserved." />

        <!-- SHIPMAI stands for :
            Secure and HIgh Performance Mobile Agents Infrastructure. -->

        <IMPLEMENTATION>
                <IMPLTYPE VALUE= "Java" />
                <LANGUAGE VALUE= "Standard UK English" />
        </IMPLEMENTATION>

        <PRICE> 10.000 C$ </PRICE>
        <WARRANTY> 5 years </WARRANTY >

        <!-- SHIPMAI needs "toto" to be installed -->

        <DEPENDENCY>
                <CODEBASE HREF= "http://www.titi.com/toto.osd" />
        </DEPENDENCY>

</SOFTPKG>
```

FIGURE 5.1: EXAMPLE OF OSD FILE

5.5. Architecture

5.5.1. System overview

MABSD is a software distribution application that is based on the deployment of agents and mobile agents. We designed three kinds of agents to perform the distribution tasks. Two of them are static and are called *PKG_INFO* (the information agent) and *PKG_DELI* (the deliverer agent). The third one is mobile and is called *PKG_INST* (the installer agent).

Besides the agents, the application architecture contains other elements. They are: the *Application User Interface* (GUI), the Web Server, the Directory Server, and the *Scheduler* (or *Organizer*) that is the heart of our application.

To give an overall idea about the philosophy of MABSD, subject to detailed description and analysis all along this chapter, we will say the following:

- The PKG-INFO agent is responsible for providing information about a software piece.
- The PKG_DELI agent is responsible for delivering the package to the client machine.
- The PKG_INST agent is responsible for installing the software in the client machine.
- The GUI is the level at which the user interacts with the application.
- The Web Server makes the GUI accessible and usable via the Internet.
- The Directory Server stores data about clients and software deliveries.
- The Scheduler (or Organizer) is the component that launches the distribution process and initializes the concerned agents at the appropriate time in the adequate order.

The following graphic (figure 5.2) gives a global view of the MABSD architecture. It represents its different components with a brief description of their respective roles and their interactions.

The next paragraphs will provide functionality details concerning the characteristics and the roles of each component apart.

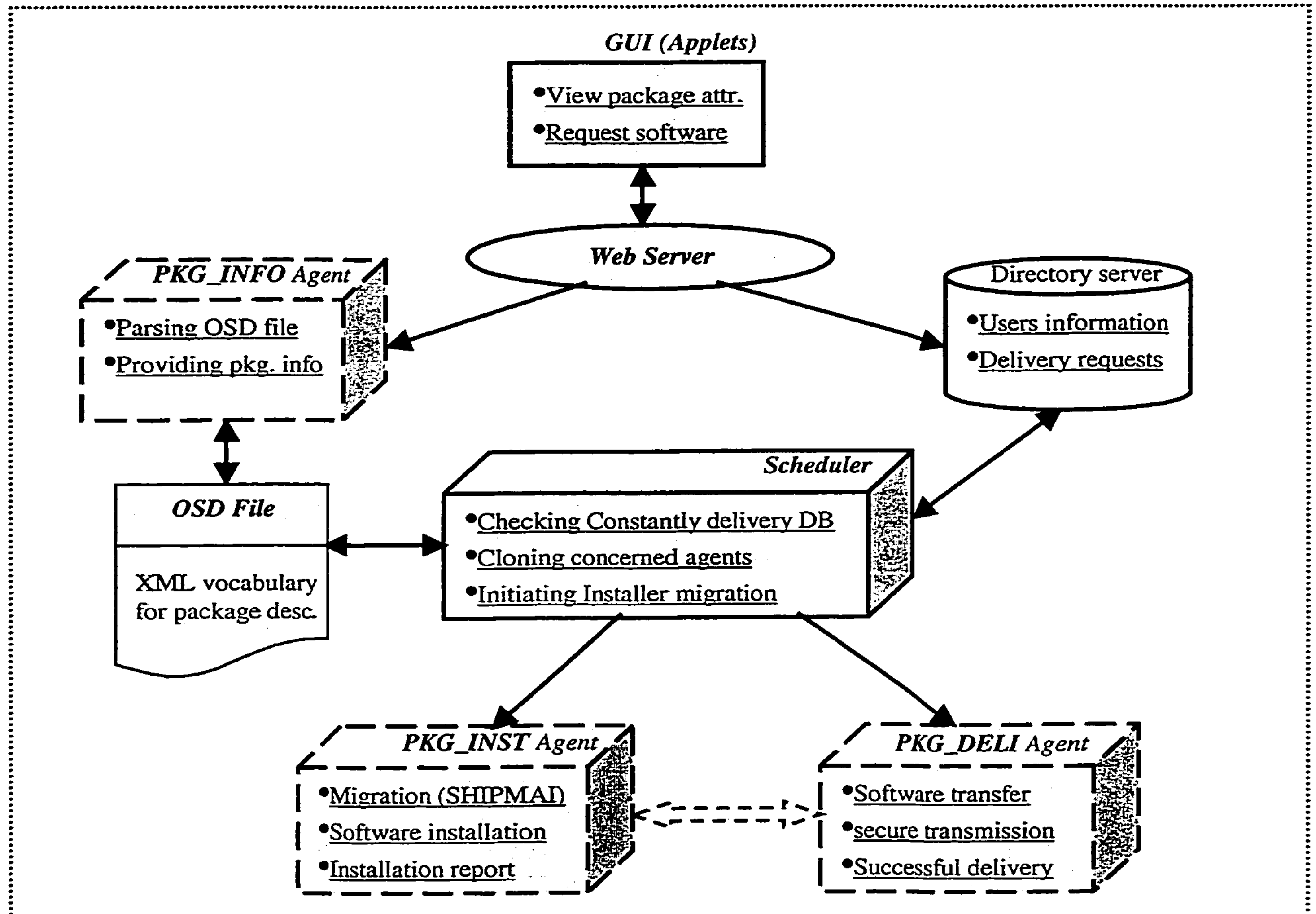


FIGURE 5.2: MABSD ARCHITECTURE

5.5.2. Information Agent

5.5.2.1. Characteristics

The PKG_INFO agent is a static agent residing at the distribution site, and destined to interact with potential clients. It is accessible to the user via the web as a Java applet.

In fact, with the fill out of the delivery request form, the interaction with the PKG_INFO agent is the only part in MABSD where the user has to intervene. The rest of the delivery and the installation process is performed automatically without user intervention.

5.5.2.2. Role

The main role of the PKG_INFO agent is to provide interested clients with all sorts of information they want to know about a given software package (name, latest version, interface language, price, required OS, CPU, memory, etc). The agent lists all package attributes, and supplies users with the accurate value(s) of the attribute(s) they asked for.

To do this, we associate an OSD file with each software package. The information agent will parse the OSD file and extract the desired data to present it to the client. Also, in case the package has dependencies on other software components, PKG_INFO informs the clients so that they can see if they have already the components installed, or whether they need to order them also. In the latter case, the clients are directed to the corresponding information agents.

If the package information change, we only need to update the OSD file (that is in text format) without having to touch the code of the PKG_INFO agent.

The intervention of the user at the PKG_INFO level is partially guided by security reasons: Clients are unlikely to trust agents for checking their machines to know their hardware and software configuration.

5.5.3. Deliverer Agent

5.5.3.1. Characteristics

The PKG_DELI agent is also a static agent residing in the distributor environment. It is a SHIPMAI agent with Java code and without interface.

The Deliverer agent is executed locally at the distribution server at the right moment to perform its task. Every PKG_DELI agent is specialized in the delivery of a specific software package.

5.5.3.2. Role

The Deliverer agent's duty consists on sending the requested software package to the client station where it should be installed. More precisely the package is sent to an Installer agent present at the client machine. The PKG_INST agent is discussed below.

The package transfer is done through a connection ensuring secure transmission of the software. Because the delivered package contains mainly executable code, the most important aspect to take care of is the data integrity and code source authentication. Hence, PKG_DELI, in collaboration with PKG_INST, uses data signature mechanism to ensure safe delivery. Encryption can also be used if it is needed.

The key force of PKG_DELI is its ability to guarantee a successful software delivery. In fact, as opposed to classical downloads, the Deliverer agent is aware at every moment of the transfer progress and can easily recover from eventual transmission problems. The delivery is not considered performed until PKG_DELI receives a positive acknowledgement from the Installer agent, which then begins its principal mission.

5.5.4. Installer Agent

5.5.4.1. Characteristics

The PKG_INST is the third and last kind of agents in MABSD. It is a mobile agent whose task is to be achieved at the remote client host. The Installer, which is also a SHIPMAI agent, is then asked to migrate from its home environment that is the distributor site.

A given PKG_INST is specialized in the installation of a specific package. To make possible several installations of the same software piece for different clients at the same time, the Installer agent is cloned at each delivery; and it is the child clone that migrates to handle the duty.

5.5.4.2. Role

As its name indicates, the PKG_INST agent is in charge of installing a specific software component in a user machine. This is very advantageous compared to classical distribution methods where the onus is on the user to install the delivered software.

The Installer agent knows everything about the package and can go through all its installation steps without any human intervention. Being a SHIPMAI agent, PKG_INST is transferred according to SHIPMAI rules for agent migration (cf. chapters 3 & 4).

In the client environment, which should be a SHIPMAI AEE, the Installer agent is executed. Its first task is to open a secure connection with the Deliverer agent in order to receive the package correctly. After that, it achieves automatically the complete setup process. Users do not have to intervene, or even to be present. They just specify the requirements in their request, and let their computer connected at the moment when the installation should take place.

Once its work is done, the clone sends a report to PKG_DELI confirming the success of the installation and containing information to keep as delivery record. Then it self-destructs.

5.5.5. Directory Server

As every user application, MABSD needs to keep important data in a persistent storage. For our application, we implemented an LDAP Directory server (see chapter 4 for LDAP overview).

The MABSD Directory server will serve to store and manage two types of information:

- *Users related data:* We store different sorts of information about registered clients under an LDAP sub-root called "*ou=clients*". Among these are personal information such as the name, the S.I.N, the address and different coordinates of the user. There are also data related to the payment mode like credit card number. Of course the client is assigned a unique login name to identify him and allow him to use the application services.

- *Deliveries related data:* When a client makes a software delivery request, all information about the request is stored in the LDAP directory server under a different sub-root called "*ou=deliveries*". The request data include the identifier of the package to deliver, the address of the machine, the specific directory where to install the software, and the date and exact time when the operation should be performed. After the delivery and the installation take place, mission reports are to be stored in order to keep record of the operation outcomes.

The MABSD Directory Server is accessed by different application entities via LDAP Java APIs. The different interactions between the MABSD components will be clearly visible when we will describe the application scenario further in this chapter.

5.5.6. Graphical User Interface

The Graphical User Interface is meant to facilitate the interaction between the user and the application. It consists of a set of applets accessible via Internet through an HTTP server hosting

the application web site. This interface was implemented using the *Java.swing* and *Java.awt* packages of the standard *JDK1.2*. GUI screens are displayed in the application prototype section.

The MABSD's GUI offers the client a visual and easy-to-use interface in order to:

- Register as an MABSD client, and provide the requested information.
- Access the application web site subject to a *login / password* authentication.
- View the software packages available for automatic distribution.
- Interact with PKG_INFO agents to know package attributes and requirements.
- Submit a software delivery request for the package he/she is interested in.

5.5.7. Scheduler

The MABSD Scheduler or Organizer is an important component that plays a primary role in the functioning of our distribution application.

Its implementation consists of a Java program playing the role of a DAEMON server.

As mentioned before, the clients' requests are stored in the application Directory server. The permanent task of the Scheduler is to keep continuously checking the directory for requests whose delivery time coincides with the current time. In MABSD, the user can specify a delivery time down to the minute level; that is why the Organizer has to poll the directory every minute to be sure not to miss any eventual request.

The Scheduler is also responsible for launching Deliverer and Installer agents at the appropriate time to perform their duties. In case of inter-dependant installations, the Scheduler

organizes the intervention of the concerned agents in the adequate order, so that the setup processes would not interfere and cause problems.

Another task of the Scheduler consists on cloning the PKG_INST agent that should migrate to client environment in order to perform the software installation. For this, it used the SHIPMAI cloning service described before (cf. chapter 3).

5.6. Class diagram

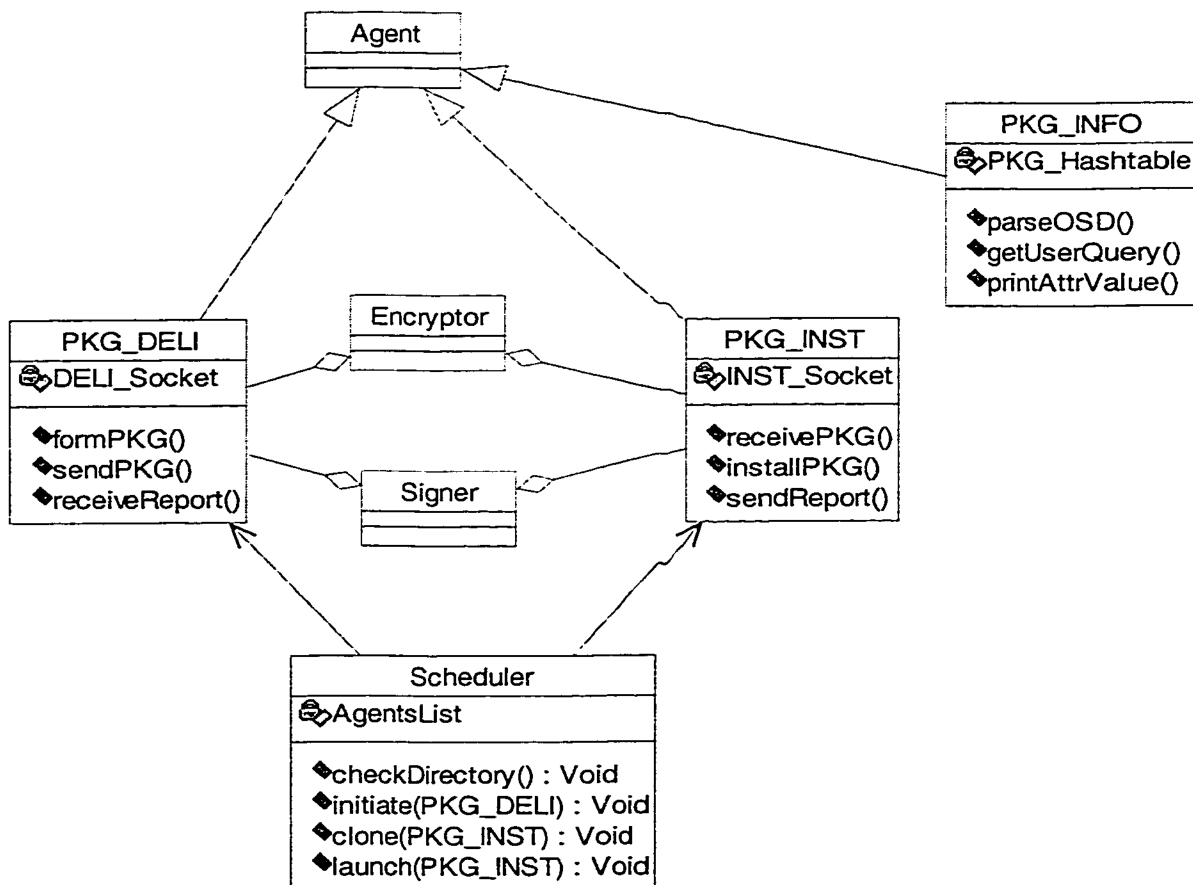


FIGURE 5.3: UML VIEW OF THE MABSD CLASS DIAGRAM

5.7. MABSD & SHIPMAI

5.7.1. Application scenario

A visual illustration of the MABSD scenario is given by the following drawing (figure 5.4), followed by a text paragraph detailing the chronological sequence of events.

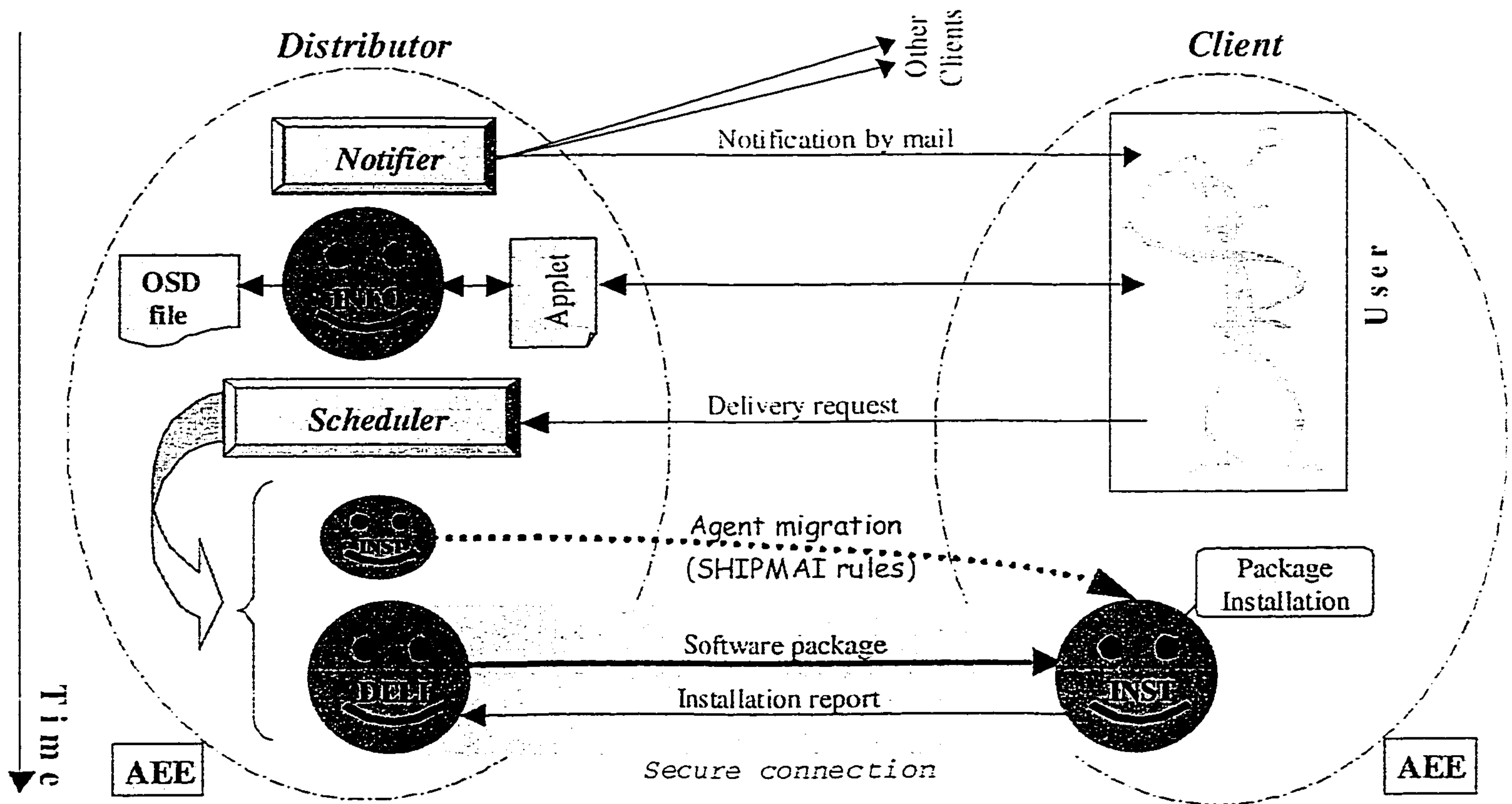


FIGURE 5.4: MABSD APPLICATION SCENARIO

Whenever the distributor site has new software tools or new versions, it automatically notifies all its clients by email. Remember that client information is kept in the Directory Server. This notification advises them to access the application web site and interact with PKG_INFO agents to know everything about the packages they might be interested in.

If a client wants to acquire a specific software package, he/she submits a query via the application GUI. The request information are then stored in the delivery directory. Every minute, the Scheduler checks this directory to look for deliveries that should be processed at the time being; in which case it programs the concerned agents accordingly.

To illustrate how MABSD works in the scope of the SHIPMAI platform, we will continue the scenario description showing how the PKG_DELI and the PKG_INST agents act under the infrastructure rules.

For generality reasons, we suppose that the distributor and the client environments are two AEEs belonging to two distinct domains in the agent-space, and thus controlled by two different ADCs. Let us suppose also that the distribution domain is managed by ADC1, and the client domain by ADC2.

When it is time for the delivery, the Scheduler initiates the execution of the Deliverer agent, which then keep listening at a server socket for the Installer connection. The Scheduler also uses SHIPMAI services to clone the Installer agent. Let us call the clone agent: PKG_INST:1.

A migration request is then sent to ADC1, which enters in negotiation with ADC2 to discuss the security measures and the resources allocation before the agent transfer. Once the agreement reached, ADC1 gets PKG_INST from the distributor AEE and send it to ADC2 through a high secure (SSL) link. Then ADC2 will be in charge of forwarding the agent to the customer environment where it will perform its duty.

After being received in the client host, PKG_INST:1 is executed by the AEE. It then opens a connection with PKG_DELI, which was listening through its server socket. The software package transmission takes place between the two agents. After completing the setup operation

and sending an installation report to the Deliverer, the cloned Installer agent is self-destructed at the client machine. The next picture (figure 5.5) is formed by reassembling execution messages displayed during a demo of the MABSD application.

These messages, presented in chronological order, describe the events related to the use of the Installer agent; and particularly how the migration of this mobile agent is performed in the scope of the SHIPMAI platform.

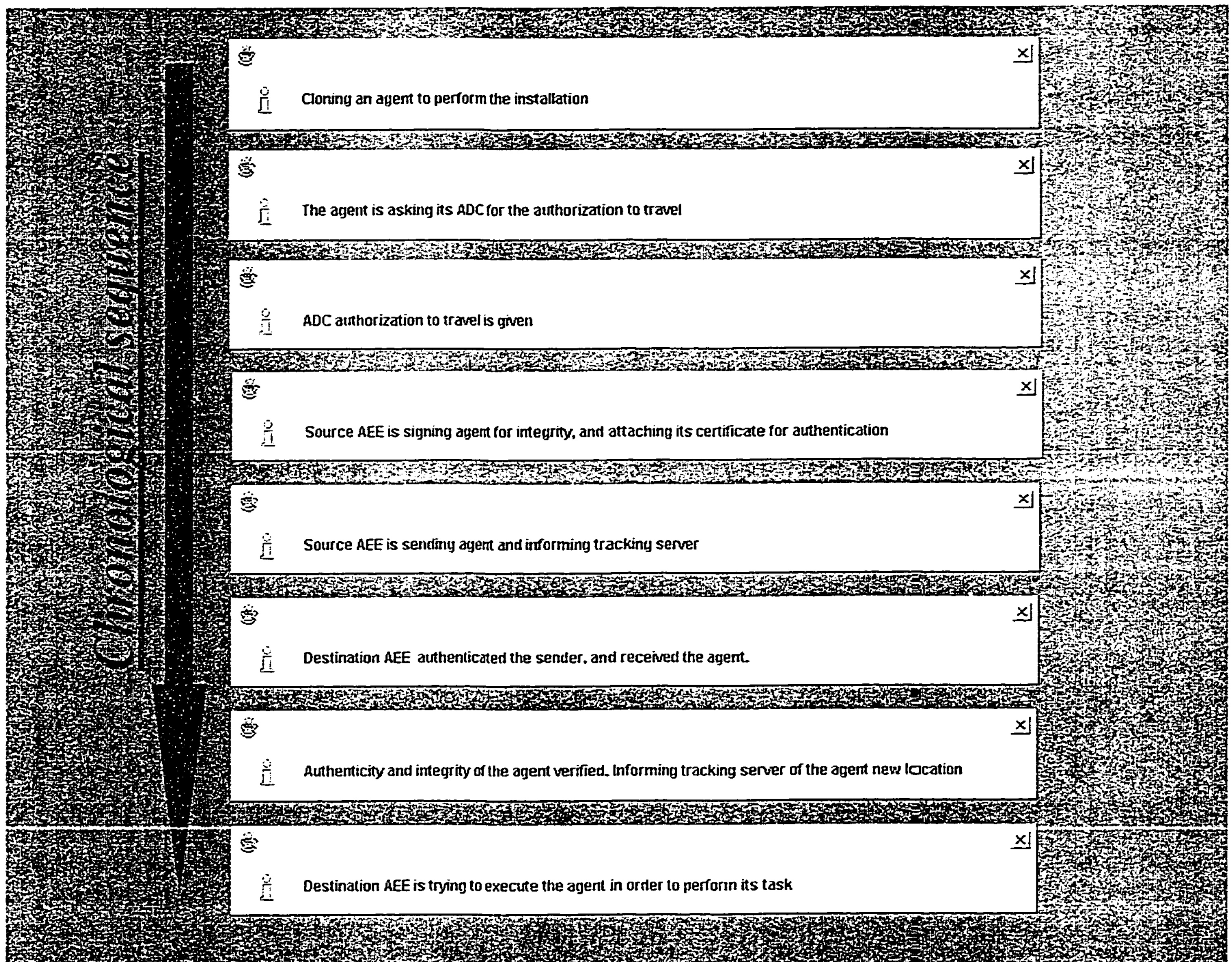


FIGURE 5.5: INSTALLER_AGENT MIGRATION STEPS

5.7.2. SHIPMAI benefits

We believe that the use of the agent technology has been beneficial for our software distribution application. Particularly, the deployment of MABSD over our SHIPMAI platform offers several advantages and interesting aspects.

Among those benefits, we can cite:

- Saving time and money relatively to classic methods by CDs or floppies.

- Assuring successful delivery and avoiding classical downloading problems by agents' cooperation (here between Deliverer and Installer agents).

- The customers have only to choose the software package that interests them and suits their system configuration. The agents will take over for the rest without the user intervention. The clients can, in fact, choose an appropriate delivery time when they are sure of not using their computers. This will avoid them being disturbed during their work by the installation process.

- SHIPMAI ensures a high security level for the agent migration (cf. security sections in chapters 3 & 4). The package transfer is also guaranteed to be safe because of the collaboration of two kit-specialized agents.

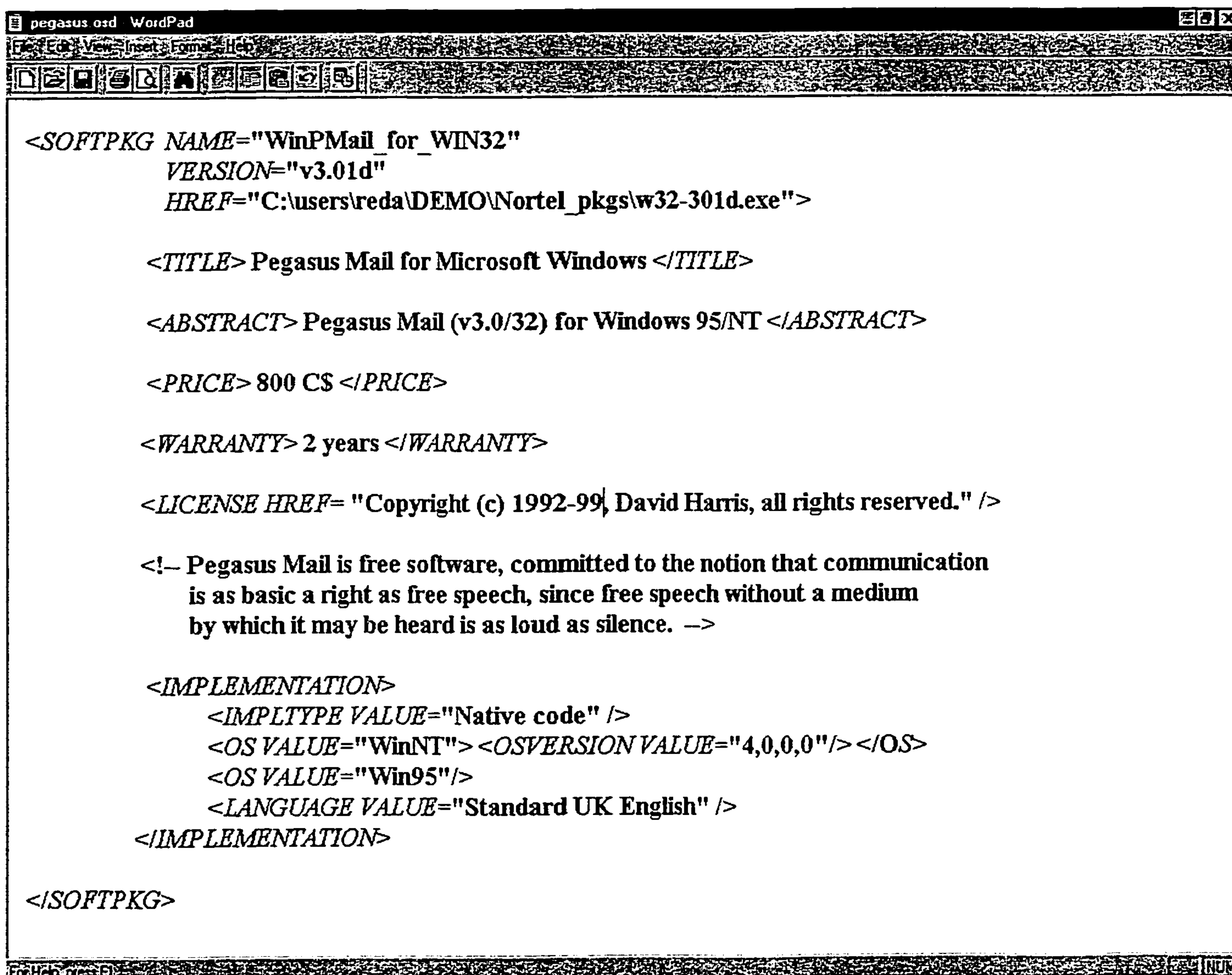
- The centralized management of the ADC allows better reusability of agents and fault tolerance in case of agent lost or failure.

Beside all of that, the development of agent-based application is required in order to get experience with the use of the agent paradigm. This will help the relatively recent MA technology to reach higher degree of maturity.

5.8. Prototype

5.8.1. Package OSD file

The following scheme (figure 5.6) shows the content of an OSD file (*pegasus.osd*) describing the attributes of a package distributed by the MABSD prototype. This is the file used by the Information agent to get the package attributes and present them to the client upon request (see the GUI screen in next page).



```
<SOFTPKG NAME="WinPMail_for_WIN32"
         VERSION="v3.01d"
         HREF="C:\users\reda\DEMO\Nortel_pkgs\w32-301d.exe">

    <TITLE> Pegasus Mail for Microsoft Windows </TITLE>

    <ABSTRACT> Pegasus Mail (v3.0/32) for Windows 95/NT </ABSTRACT>

    <PRICE> 800 C$ </PRICE>

    <WARRANTY> 2 years </WARRANTY>

    <LICENSE HREF= "Copyright (c) 1992-99, David Harris, all rights reserved." />

    <!-- Pegasus Mail is free software, committed to the notion that communication
         is as basic a right as free speech, since free speech without a medium
         by which it may be heard is as loud as silence. -->

    <IMPLEMENTATION>
        <IMPLTYPE VALUE="Native code" />
        <OS VALUE="WinNT"> <OSVERSION VALUE="4,0,0,0"/> </OS>
        <OS VALUE="Win95"/>
        <LANGUAGE VALUE="Standard UK English" />
    </IMPLEMENTATION>

</SOFTPKG>
```

FIGURE 5.6: PROTOTYPE PACKAGE OSD FILE

5.8.2. Viewing software attributes

The GUI screen below (figure 5.7) represents the MABSD interface allowing the user to select a specific software package and interact with the PKG_INFO agent in order to be informed about the characteristics of the software. The data shown in the "*Information Detail*" text area is directly retrieved from the package OSD file (shown in the previous page).

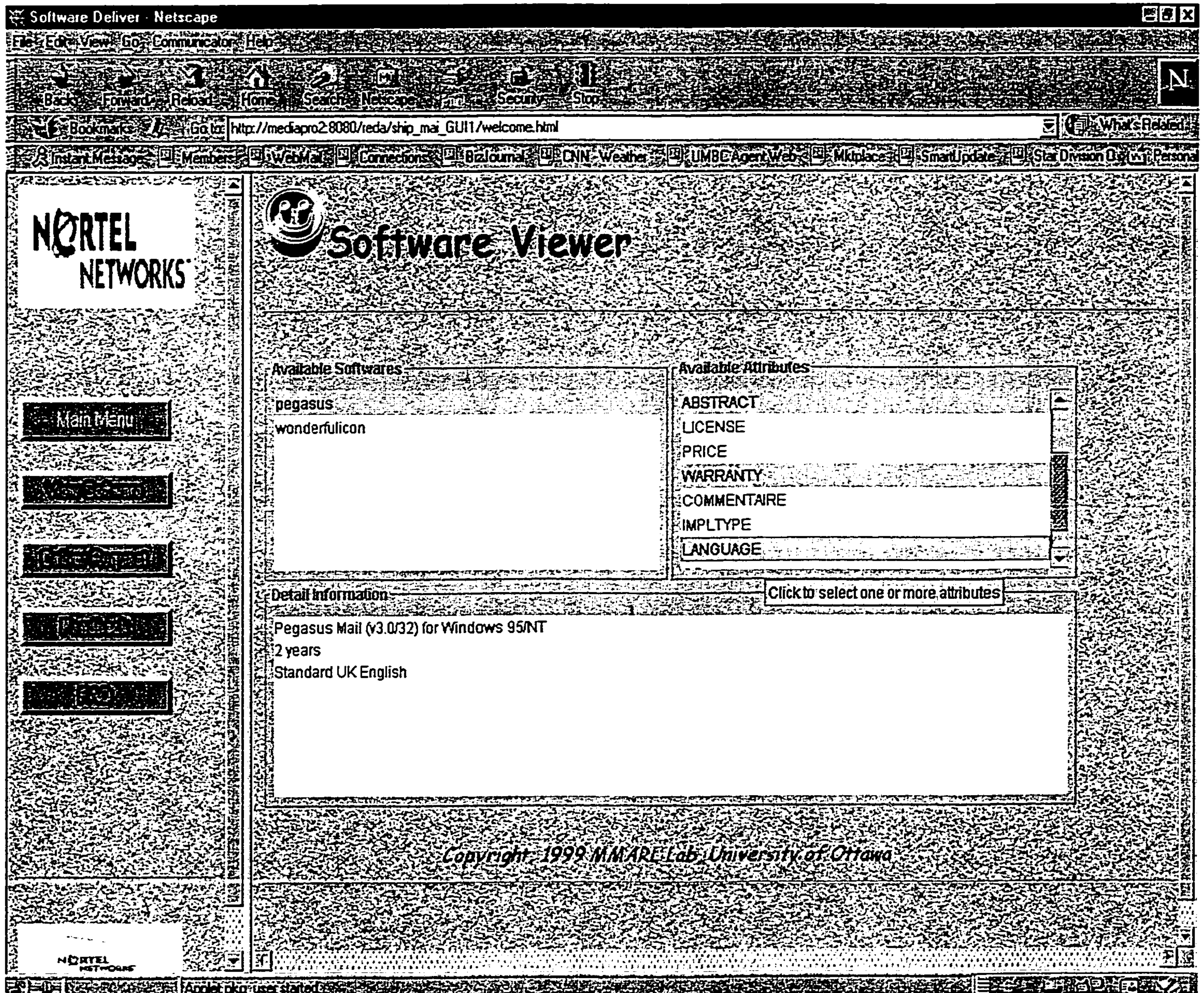


FIGURE 5.7: GUI SCREEN FOR "USER / INFORMATION_AGENT" INTERACTION

5.8.3. Delivery request

The next screen figure (figure 5.8) shows the web form used by MABSD client to submit delivery request for the packages they are interested in. As shown, the user specifies, among others, its machine URL address and the exact time where the delivery should be handled. Once the form submitted, all the request information are stored in the Directory Server.

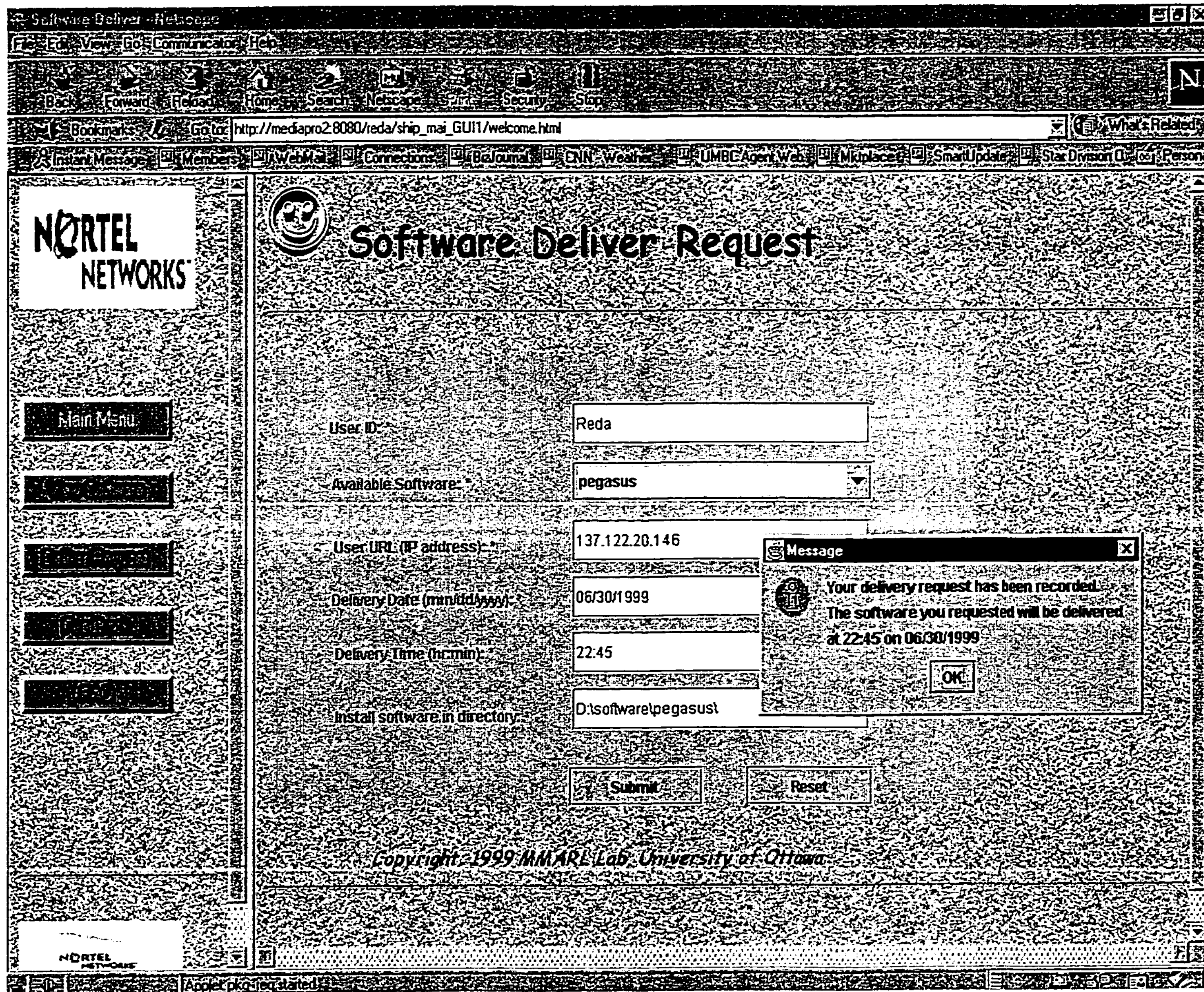


FIGURE 5.8: GUI SCREEN FOR SOFTWARE DELIVERY REQUEST

5.8.4. Launching delivery process

The Scheduler, as illustrated in the following screen catch (figure 5.9), is permanently checking the directory for eventual requests to process. When the time of the delivery specified by the client in the web form arrives (that is 22:45), the Scheduler launches the processing by first executing the Deliverer agent, and then initiates the migration of cloned Installer agent.

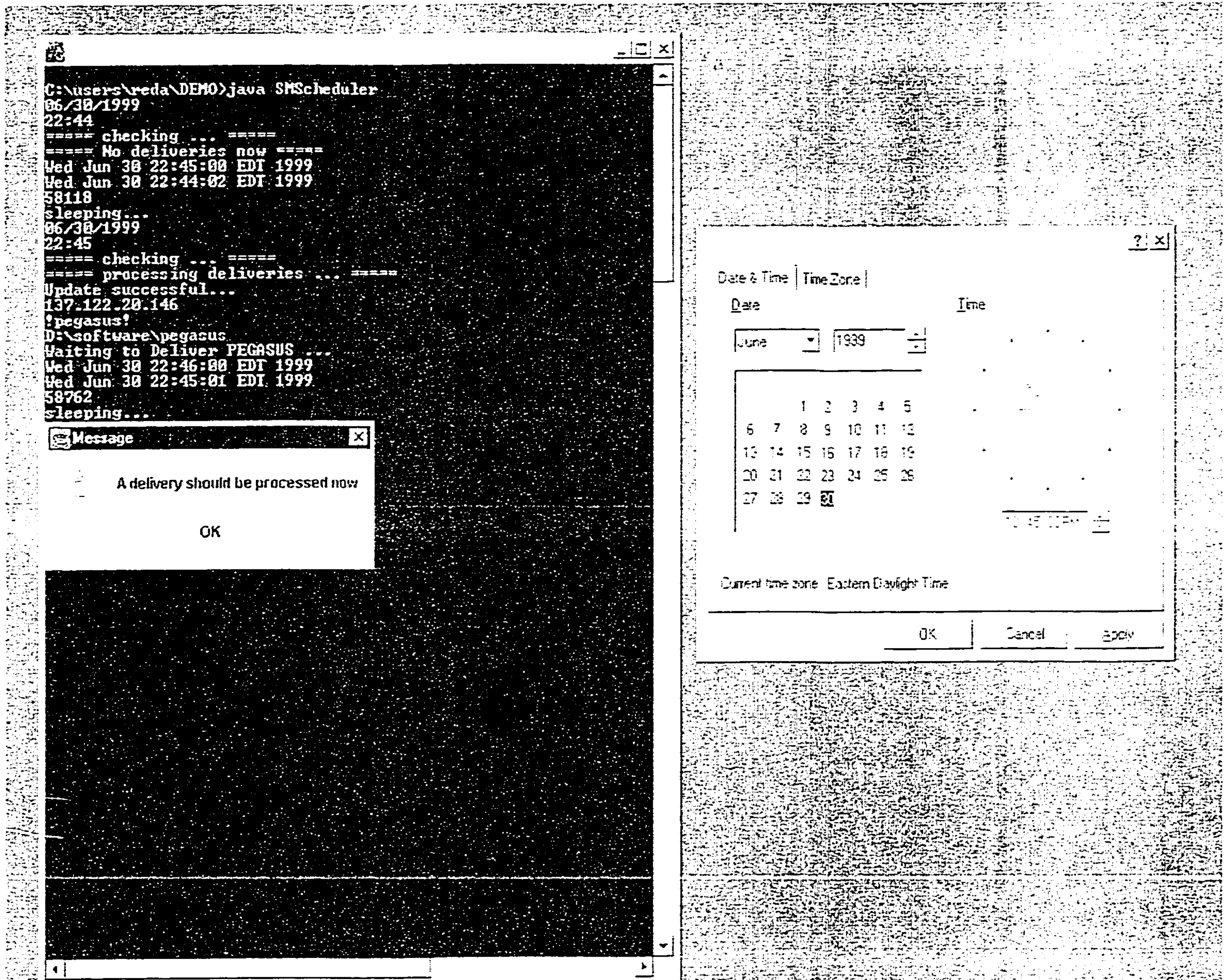


FIGURE 5.9: GUI SCREEN SHOWING DELIVERY LAUNCHING

Chapter 6

6. POLICIES-BASED SHIPMAI

6.1 Introduction

The work presented in this chapter deals with the use of policies as functional basis for the SHIPMAI platform. Brief definitions concerning policies will be given; they will be adopted all along the chapter. Then, after specifying our motivations, we will describe in detail our strategy for the integration of policies in the infrastructure components and in the agent objects as well.

6.2. Brief definitions

In last years, distributed systems witnessed a continual increase in their size and complexity. As a result, tasks related to system management grow to be more and more difficult, thus tending to become highly automated.

This automation has been accompanied by an intense research activity concerning management polices. Particularly, researchers in the imperial college of London have had a great interest in this topic and have written many papers about policies. Their definitions and specifications are adopted all along this chapter.

Policies are defined in [55] as "*the plans of an organization to meet its goals*". They are intended to provide positive guidance about these goals and how they should be met. Researchers classify policies under two categories:

- ***Obligation policies:*** Specify what a manager must or must not do. They are responsible for motivating actions to be taken when a particular events occur.

- ***Authorization policies:*** Specify what a manager is permitted or not permitted to do. They are responsible for empowering actions so that they can take effect.

This distinction stipulates that an action cannot be carried out unless there is an obligation policy that motivates it, and an authorization policy that allows it to be executed.

Research work have also focused on defining an adequate representation for policies, so that they can be modeled as objects that can be queried, manipulated, and used within computer systems. In [56], authors have defined the following attributes for a policy object:

- *Identifier:* A string identifying the policy and referring to it.
- *Mode:* **O+**, **O-**, **A+**, or **A-** (stands for positive/negative obligation/authorization).
- *Trigger:* The event provoking an obligation (authorizations do not need triggers).
- *Action (goal):* Operation(s) that must be done or are permitted to be done.
- *Subject:* Object(s) obliged or authorized to carry out the action.
- *Target:* Object(s) on which the policy action is performed.
- *Constraints:* Conditions for the policy to take effect (optional).
- *Related policies:* parents, children, or references for this policy (optional).

With these attributes, the notation for representing a policy would be:

Identifier Mode [Trigger] Subject '{ ' Action ' }' Target [Constraints] [Related policies]

For the examples of policies given in this chapter, we will not mention the identifier attribute since it is not important for our purely illustrative purposes.

By modeling policies as objects with defined attributes, it becomes easy to keep them in a persistent storage (a directory or a database). They could also be represented as runtime objects that can be used by computer programs.

6.3. Motivation

The importance of structuring policies for distributed systems is rapidly growing. However, policies, up to now, are used for pure system management purposes only.

Our contribution in this work is to extend the use of policies for more general aims. Concretely, our goal is to make policies the functioning basis of our mobile agent platform (SHIPMAI). In other words, we want to restructure SHIPMAI components so that their functionality and services would be entirely driven and controlled by policy objects.

We are motivated by the following idea: since policies can enforce and regulate management tasks, then there is no reasons they cannot regulate server and program tasks as well.

By basing the entire infrastructure on policies, we could easily modify the way the platform components interact with each other. Particularly, assigning policies to mobile agents will allow us to dynamically change their behavior by simply changing their policies. This would be somehow equivalent to building "intelligent" agents, but without having recourse to Artificial Intelligence methods that are often complex and difficult to implement.

6.4. Orientation

As aforementioned, our goal is to restructure our SHIPMAI platform by basing the majority of its functionality upon policies. That is, every action to be performed has to be driven by an obligation policy and allowed by an authorization policy. The desired policy-based management concerns all the system entities. We will classify these entities into two different categories:

The first covers the infrastructure component blocks, which are the User Management Service (UMS), ADC, AEE, and the Administrative Center (AC). The User Application Interface is out of concern since it only offers graphical interface for user interaction without any kind of control needing to be handled by policies. The second category concerns mobile agents, which, because of their mobility, require a different strategy for the integration of policies.

The rest of this section will detail our design for adapting policies to the infrastructure entities and the mobile agents. It is worth saying that this policy integration is designed to control the platform behavior and services independently from any kind of applications that could be built on top of SHIPMAI.

6.5. Policies for Infrastructure Components

6.5.1. Architecture

For every infrastructure component (i.e. UMS, ADC, AEE, and AC), and depending on its responsibilities, we assign a certain set of policies that will guide its functional behavior. The schema that follows (figure 6.1) refreshes our memories about the SHIPMAI architecture, and shows pictorially the affectation of policies to the infrastructure components.

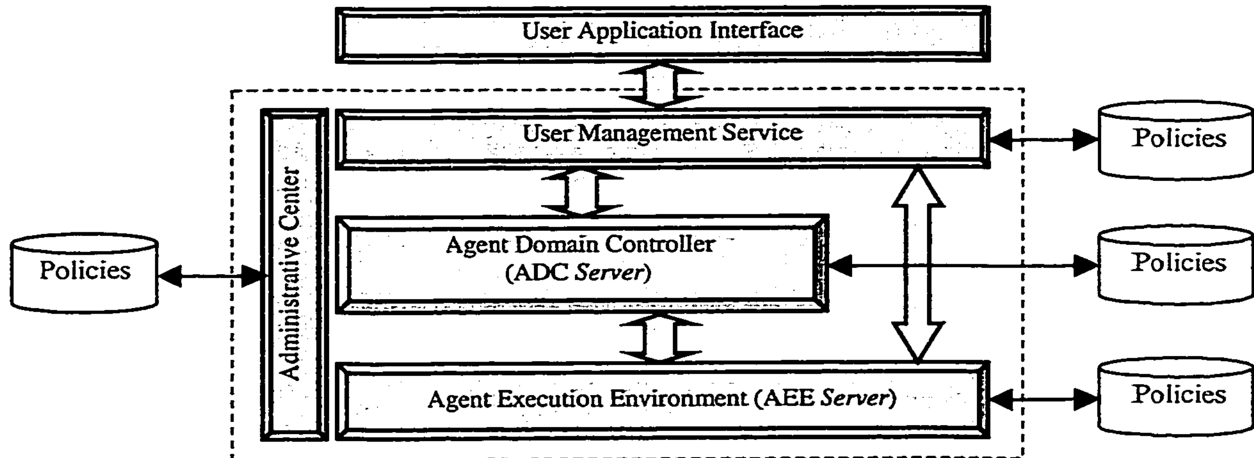


FIGURE 6.1: SHIPMAI POLICIES ASSIGNATION

In what follows, we will specify the kind of policies adopted by each component in order to help it control its actions and make its decisions. We adopt the most common categorization of policies, classifying each component's policies under two types: Obligation policies and Authorization policies.

6.5.2. User Management Service (UMS)

Let us recall that the UMS is the layer component responsible for managing system users. UMS tasks consist mainly of controlling customers operations. This includes for example validating users requests before forwarding them to the concerned entity responsible for answering them.

6.5.2.1. Obligations

Obligation policies at the level of the UMS motivate it to perform a specific action at the reception of a user request. Therefore, the trigger of such policy will always be a user request and its subject will always be the UMS. An example of an UMS obligation policy is :

*O+ **on** ur:user-request UMS { forward to ADC } ur **when** ur.type ∈ {AgCreate, AgLocate}*

which means that, at the reception of a user request *(trigger)*, the UMS *(subject)* should *(modality)* forward *(action)* the request *(target)* to the ADC, if the type of the request is agent creation or agent location *(constraints)*.

6.5.2.2. Authorizations

UMS authorization policies check the legitimacy of users requests and decide whether they should be allowed or denied. Hence, the subjects of these policies are always users.

Suppose the UMS has received a request for agent creation, the policy aforementioned motivates it to forward it to the ADC. However it would not do so unless there is an authorization policy that empowers the action to take place. Such policy would be of the form:

*A+ u:user { create } agents **when** u∈ {MMARL members}*

This policy states that only MMARL members are authorized to create agents. The UMS will then forward the former request to the ADC only if the requesting user is a MMARL member. Other UMS authorization policies allow or forbid users to/from logging on to the system or performing operations such as locating agents, getting their work results, terminating them, etc.

The following policy authorizes a user to perform certain operations on an agent if this agent belongs to the user ACL (Agent Control List).

*A+ u:user { locate, freeze,terminate } a:agent **when** a∈ u.ACL*

6.5.3. ADC and AEE

The ADC and the AEE are the heart components of SHIPMAI. They are the servers that take care of all agent-related issues and provide agents with all features they need to deploy. Since both ADC and AEE are agent service providers, their functioning principles are the same. Only the services offered by each one of them are different. Thus, we mix between the two components when talking about their policy strategy.

Our intention is to drive the services offered by ADCs and AEEs by policies. That is, we define authorizations and obligations that would be queried in order for our servers to make the right decision and carry out the appropriate action.

6.5.3.1. Obligations

As explained in chapters 3 and 4, the communication framework of SHIPMAI is based on a uniform messaging mechanism. ADC and AEE are thus considered as message-reactive; which means that every action they perform are engaged by a received message.

Hence, we can already foresee that for ADCs or AEEs obligation policies, the triggers will necessarily be SHIPMAI messages. Also, being in charge of receiving messages, the servers' *Message Handlers* will constitute the policies subjects.

Messages handlers (MH) are also responsible for invoking dynamically appropriate methods of objects. The obligation policies will then specify what method should be invoked on which object in order to provide the required service. The following is a typical ADC / AEE obligation policy.

***O+ on** mr:message-received MH { invoke method:met} objet:o* ***when** [o∈ mr.RegisterdObjects **and** met==mr.name]*

This policy illustrates the SHIPMAI dynamic method invocation stating that the method with the same message name should be invoked on the object registered for this kind of messages.

6.5.3.2. Authorizations

The message handler does not tamper deeply with the message internal structure. It only extracts the message name to know which method it has to invoke. The other message attributes are extracted at the level of the ADC or AEE object on which the method is invoked. Thus the decision of whether to provide or to deny the service requested via the message is to be made at the ADC or AEE level. For such decision, the server consults its authorization policies.

Agents are usually subjects or targets of these authorization policies. This is because they are either the entities requesting services or the ones concerned by a user or another agent request. As for policies actions, they can be of different nature depending on the message request. The following ADC authorization policy allows the domain native agents to send inter-domain messages without any constraints; which may not be the case for visitor agents.

A+ native agent { send } inter-domain messages

Next is an authorization policy allowing the AEE server to execute a visitor agent only if the server workload is less than 75%.

A+ AEE { execute } visitor agent ***when*** *workload < 0.75*

The figure below (figure 6.2) illustrates how ADC / AEE actions are motivated and empowered by consulting obligation and authorization policies respectively.

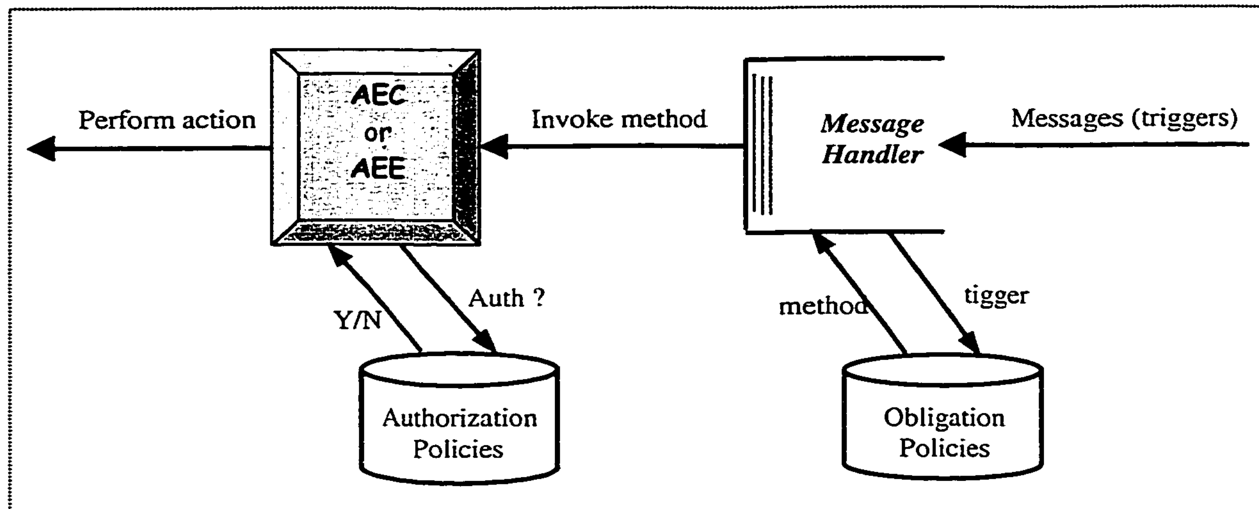


FIGURE 6.2: ADC/AEE POLICY STRATEGY

6.5.4. Administrative Centre (AC)

The AC is the administrator of a SHIPMAI domain. It has full control of all servers, system settings, and infrastructure components whose functionality, as described in the above paragraphs, is now driven by policies. Therefore, the new duty of the AC will be the management of the policies of the other domain entities. This management will be handled by AC ***meta-policies***. It stipulates that AC policies will consist of "*policies about policies*".

The Administrative centre consists on a set of controlling applications used by administrators (human agents) to take care of the good functioning of domain servers. Therefore, AC meta-policies have these human agents as subjects, and domain policies as targets. Their actions are operations that can be performed on policies. We restrict ourselves to four operations: add, remove, enable and disable a policy.

6.5.4.1. Obligations

AC obligations meta-policies will motivate actions to be taken by the administrators when special events occur in the system, such as server crash, network congestion, or host overload.

For example, if a host server is simultaneously executing a great number of agents, it might witness some overload. In this case, the system performance manager must disable some AEE authorization policies that give agents access to critical resources constituting eventual server bottlenecks. Such action would be enforced by a meta-policy of the form:

O+ ***on*** *host overload* *Performance manager* *{ disable }* *some AEE auth. policies*

6.5.4.2. Authorizations

As an action motivated by an obligation policy cannot take effect without an authorization policy that empowered it, the AC should have authorization meta-policies that allow administrators to manipulate domain policies.

Hence, the performance manager cannot execute the action required by the above obligation unless there is an authorization meta-policy stating:

A+ *Performance manager* *{ enable, disable }* *AEE auth. policies*

6.6. Policies for Mobile Agents

6.6.1. Vision

SHIPMAI mobile agents are entities that roam agent domains to perform tasks and require services. Our vision consists of controlling the behavior of agents by means of policies. We want these policies to be carried by agents so that they can locally consult them at any time. This will certainly weigh down agents but will advantageously reduce the time and network traffic of remote policy consultation that would be necessary if the policies are stored outside the agent.

We want also to be able to modify remotely agent policies, thus changing dynamically the behavior of agents and the way they react to events. By this, we can attain some agent intelligence, and made policies a good and simple alternative to the use AI typical techniques.

Agents' actions can be classified under two types: Actions performed in the scope of its mission. These are application-specific and are not of interest to us. The other types of action are those related to the platform, that is the interactions with ADCs and AEEs. These are the kind of actions we aim to subject to policies.

Also, since these latter actions are always triggered by some event (such as agent migration, ADC directive, etc), agents would have to carry obligation policies that should influence these actions. On the other hand, agents do not need to carry authorization policies because they are requesters of services; which makes them subjects for authorization policies that are attached to target objects (services providers). Attaching authorizations to targets is related to the principle of hiding policies for security reasons or negotiation purposes. It means that an AEE, for example, would not inform an agent in advance about the resources it is allowed or forbidden to access. The grant or denial is revealed only when the agent asks for the resource.

6.6.2. Obligation policies

SHIPMAI agents should then carry obligation policies directing their acts anytime and anywhere, either in their home domains or in foreign territories. We divide agent's obligation policies into native (or permanent) policies, and visiting (or temporary) policies.

6.6.2.1. Native policies (NP)

Native policies are policies assigned by the home ADC to the agent once it is created and registered to the domain, hence their appellation. We also call them permanent policies because

they stick to the agent during its whole life span. These NP oblige agents to respect the platform rules, react accordingly to the constraints imposed by SHIPMAI, and specially regulate its interactions with its domain authorities.

For example, a native policy would force agents, at each migration, to send their work results to the ADC to be stored persistently for fault tolerance purposes. Another permanent policy would induce agents, when received by a host, to present their certificates to the AEE server.

O+ on migration Agent { send to home ADC } work results

O+ on reception by host Agent { present to AEE } certificate

6.6.2.2. Visiting policies (VP)

Visiting policies are attached to agents only when they are visiting foreign domain servers. They are also called temporary policies since they are carried and consulted by agents only as long as they are in external domains.

VP are assigned by a foreign ADC to agents before allowing them to enter its domain. They oblige agents to respect any particular domain laws in force. A possible ADC policy might require visitor agents not to stay in its domain after 9:00 PM.

O+ at [9:00 PM] Agent { leave } domain

Of course, the examples of policies we present in this chapter are often high-level policies with abstract directives or goals. Each policy has to be refined to one or more low-level policies with concrete actions that could be performed by software entities.

Since visiting policies enforce the particular constraints of a SHIPMAI domain, the ADC removes them from agents when they are leaving its controlled area (figure 6.3).

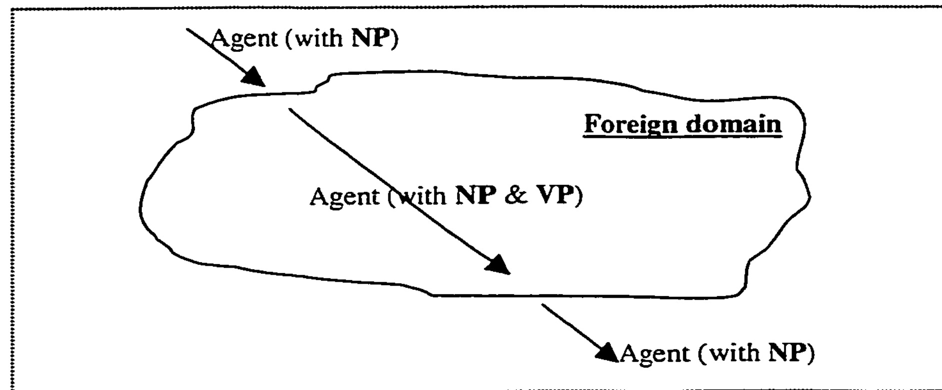


FIGURE 6.3: NATIVE AND VISITING AGENT POLICIES

6.6.3. Authorization meta-policies

We have stated that agents would not need to carry authorization policies. They will, nonetheless, carry authorization meta-policies to protect their other policies from unauthorized manipulation, since we want to modify agents obligations for dynamic behavior changing.

Therefore, meta-policies serve to control these modifications and specify which entities are authorized / forbidden to perform which operations on agents' obligation policies (native and visiting). As mentioned before, we identified four basic operations on these policies. They can be added, removed, enabled, or disabled.

Two examples of agent authorizations meta-policies follows:

A+ Home ADC { add, remove } A native policy

A- Domain AEE { enable, disable } A visiting policy

The following figure (figure 6.4) shows the policy structure of SHIPMAI agents. Obligation policies (NP and VP) are triggered by events to specify the action the agent should make. The agent offers interfaces to manipulate these VP and NP, subject to authorization meta-policies.

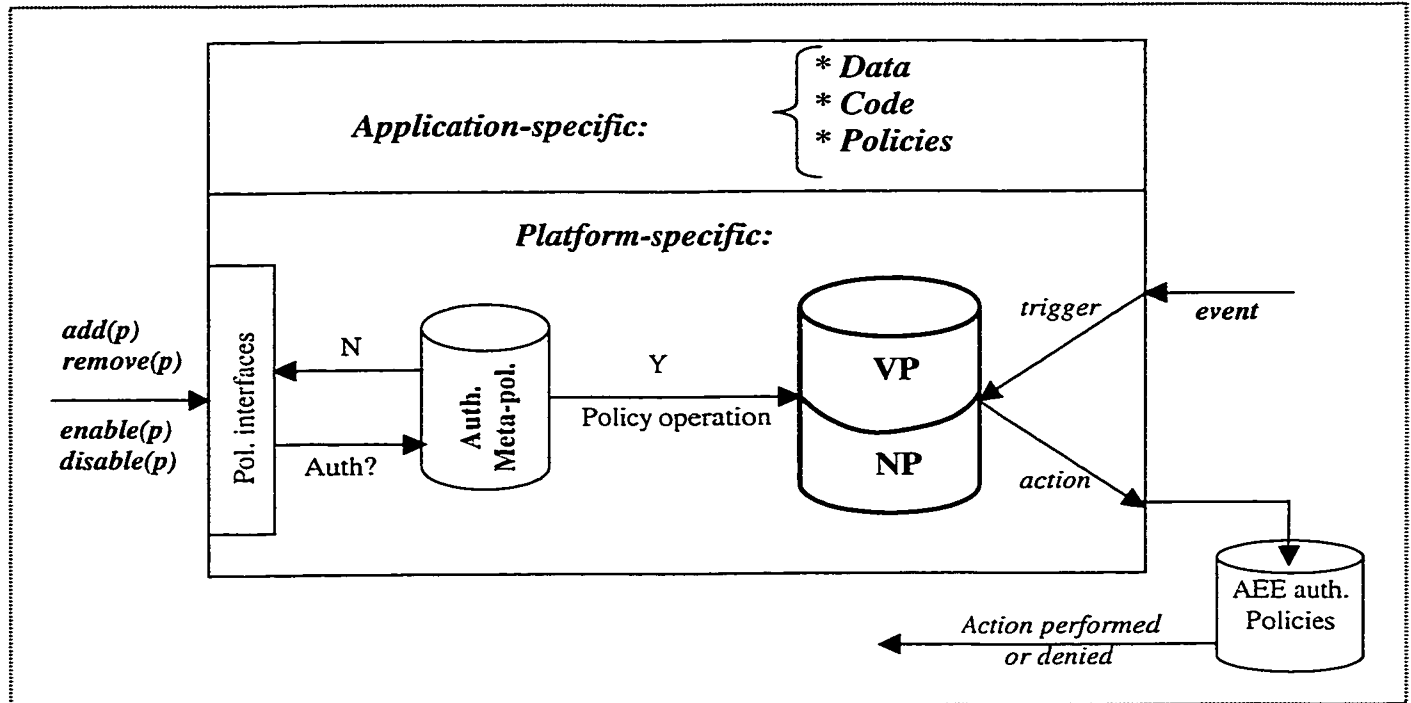


FIGURE 6.4: AGENT POLICY STRUCTURE

6.7. Policies storage

6.7.1. Policies as objects

As aforementioned in the first section of this chapter, we adopt the policy syntax defined in [56]. Thus, we will consider our policies as objects having the following attributes: *id, mode, trigger, subject, action, target, and constraints.*

Before storing SHIPMAI policies, we have to define them clearly and precisely. First, we should define high-level policies defining the goals to achieve (like the ones we gave as

examples all along this chapter). Then those high-level policies should be refined to low-level policies that would exactly specify the concrete action (invoke a method, send a message, etc) that should be performed, and the software entity to perform it.

Once the policies have been defined, they have to be stored somewhere so that they can be easily consulted and manipulated. In our implementation, we distinguish again between infrastructure components and mobile agents in the way their policies are held.

6.7.2. Infrastructure Components' policies

As these platform blocks are static, we decided to store their policies in a persistent storage such as a directory. And because they are scattered among geographically distant sites, we thought about a distributed directory service.

The choice of the tool was obvious to us. We chose LDAP for the same reasons described in chapter 4, plus there was no meaning to add the complexity of using another product. The next graphic (figure 6.5) shows the hierarchy of the SHIPMAI policies' directory.

We assigned a directory sub-root to each infrastructure component. Under each sub-root lie two branches: one for obligations and one for authorization policies.

A policy interpreter is built to deal with the consultation of policies and the execution of their actions. It is a Java class accessible for infrastructure programs. When needed, it is used to access the policy directory, extract the policies that should apply, and convert their actions to method calls that it directly invokes at runtime. This provides great flexibility for platform functionality and useful abstraction in source codes.

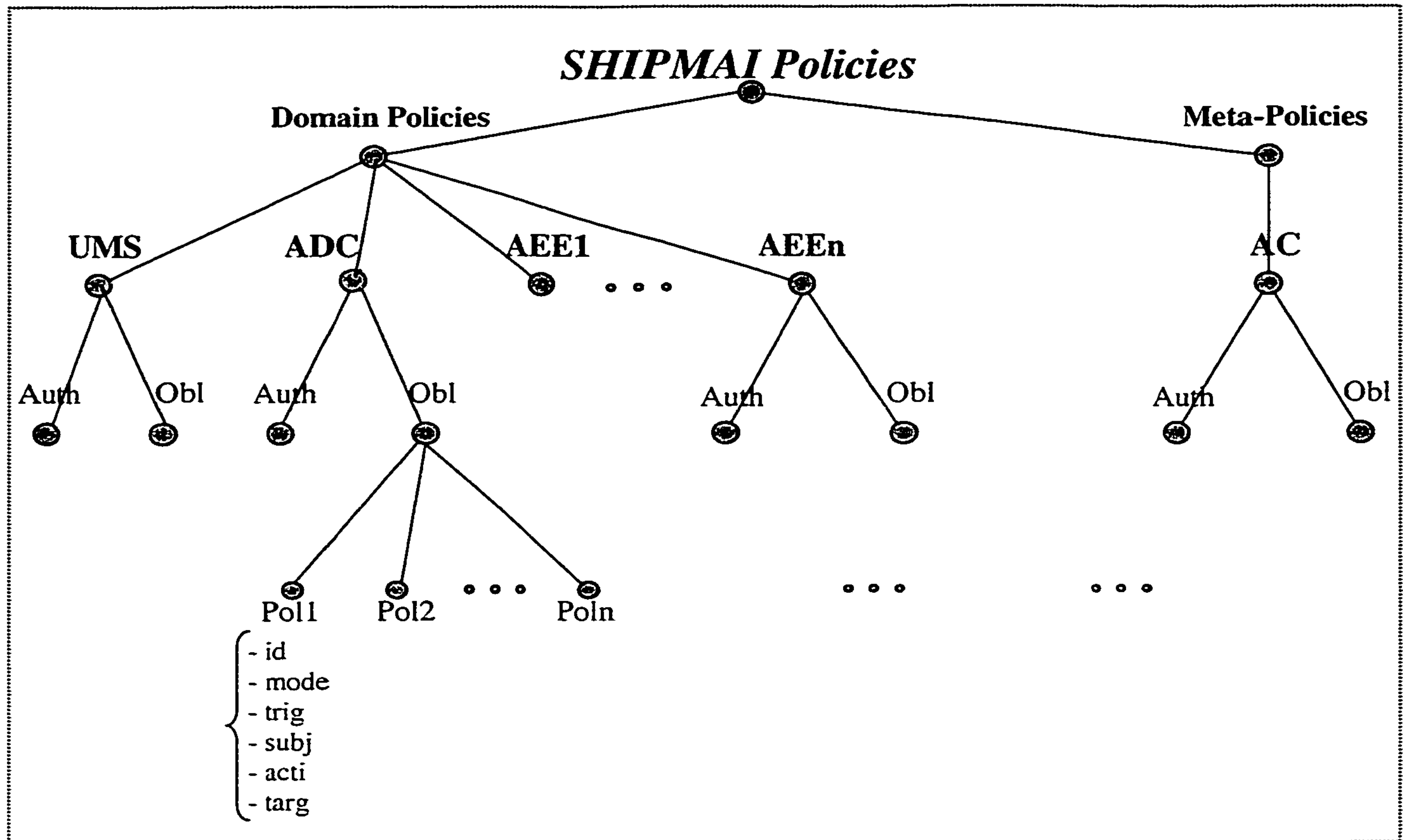


FIGURE 6.5: DIRECTORY HIERARCHY FOR SHIPMAI POLICIES

6.7.3. Mobile Agents' policies

Unlike static platform entities, storing mobile agents policies in a directory is not a good idea, precisely because of their mobility. Such conception will generate great network traffic and will be hard to manage. That is why policies should be attached to agents.

To do this, policy Java objects are aggregated to the agent Java object in the form of vectors or arrays of objects. According to our design, two vectors containing policy objects will exist inside the agent:

- One for native policies (NP given by the home ADC). This vector will be permanently part of the agent object.

- One for visiting policies (VP given by at the foreign visited ADC). It will be temporarily carried by the agent during its stay in a foreign domain.

Agent objects provide interface methods allowing authorized entities to manipulate their policy vectors. Hence, policies could be changed remotely using the SHIPMAI messaging mechanism, allowing a dynamic change of the behavior of agents.

The following class diagram (figure 6.6) shows the relation of aggregation between the agent and the policy classes.

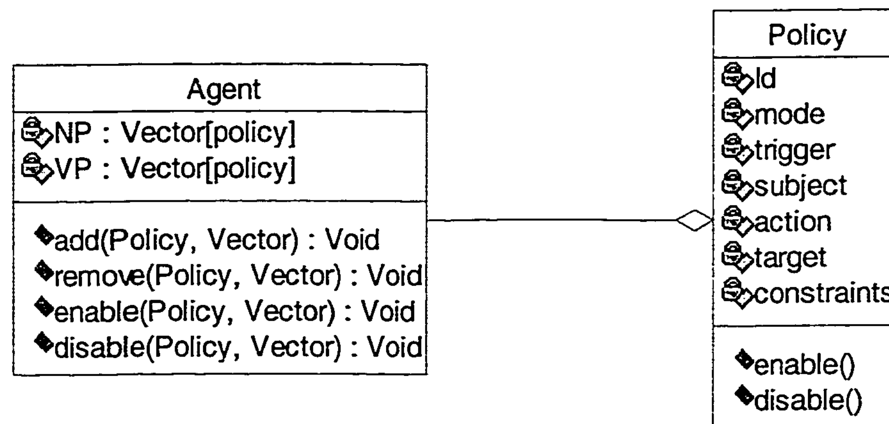


FIGURE 6.6: AGENT POLICIES CLASS DIAGRAM

Chapter 7

7. Conclusions

7.1. Summary

Although the mobile agent paradigm is the central topic investigated in this thesis, two other themes, which are software distribution and policies, have been considered and dealt with in the scope of our research work.

The bibliographic part has consisted on covering the background material in the software agent technology and analyzing the state-of-the-art research work related to agent platform design. This analysis has driven us to develop our own mobile agent infrastructure (SHIPMAI) and to design our proper agent model. The infrastructure has been experimented by the implementation of an agent-based application for automatic software distribution (MABSD). Finally, in order to improve the flexibility of SHIPMAI, policies have been integrated to the system in order to regulate the functioning of agents and platform components.

The developed system creates a reliable and secure environment suitable to the deployment of mobile agents. The offered agent services include persistence, mobility, communication, security, tracking, and cloning. The key force of SHIPMAI is the fact that it divides the network into separate and controlled agent domains. Such design has proved to be beneficial regarding many aspects among which security is the most important. Intranets have practically proved that private domains greatly simplify the security strategy and make it more efficient.

The infrastructure also introduces a smart distribution of agent services. It assigns critical tasks, such as security, control, and persistence, to Agent Domain Controllers (ADCs); and let Agent Execution Environments (AEEs) focus more on providing agents with resources helping them to achieve their work. This distribution lessens the burdens of agent servers, provides a better fault tolerance, and enhances the robustness of the system.

The integration of policies was meant as an enhancement for SHIPMAI. Policies are used to control the functioning of platform entities, thus helping for great system flexibility. As for the injection of policies into agents, it aims to simplify their remote control and to ease the changing of their behavior with respect to different situations.

Mobile agents are promising in many application areas. In this thesis, experimenting them for a software distribution application was found to be of great interest. Agents can considerably help in resolving problems of automatic software delivery and installation. MABSD has also contributed to assess the ability of SHIPMAI to support interesting distributed applications.

The choice of Java as programming language is reasonably justified. Thanks to JVM, Java has become the de facto standard language for heterogeneous systems. It takes care of the code portability, thereby letting us better focus on more specific agent issues. Also, the virtual machine offers a safe environment for executing mistrusted code, and thus helps in protecting agent host servers.

7.2. Future work

Research on the topic of mobile agents, their supporting infrastructures, and their potential applications has never been as active as it is actually. Many aspects of the MA paradigm and its

domain of interest are currently the subject of further investigations. In our opinion, security is the aspect that should attract the most important part of research efforts.

In mobile agent systems, as in all open software systems, the security issue could never be considered as definitely settled. In this perspective, we have engaged further work to extend the SHIPMAI hierarchy. Although ADCs are considered as legitimate and potentially trusted authorities, one may discuss the degree of their trustworthiness. That is why we have thought of adding a higher hierarchy level, which will consist of few super-ADCs that will supervise sets of domains, and whose credibility would be universally incontestable.

Also, for more experimentation of our infrastructure, several agent-based applications using SHIPMAI as supporting platform are currently under development in our research laboratory. These applications concern areas that witness growing interest, such as electronic commerce and network management based on quality of service (QoS).

8. References

[1] R. Tkito, A. Karmouch, "SHIPMAI: a Secure and HIgh Performance Mobile Agent Infrastructure", Proc. of the IEEE Canadian Conference for Electrical and Computer Engineering, Halifax, Canada, May 2000.

[2] R. Tkito, A. Karmouch, "Towards a communicative and secure agent space", Proc. of the 20th Biennial Symposium on Communications, Kingston, Canada, May 2000.

[3] "The Agent Society".
Home page: http://www.agent.org/.

[4] V.A. Pham, A. Karmouch, "Mobile Software Agents: An overview", In IEEE Communications Magazine, 36(7), pages 26-37, July 1998.

[5] M.R. Genesereth and S.P. Ketchpel, "Software Agents", In Communications of the ACM, 37(7), pages 48-53, July 1994.

[6] J.R. Galliers, "A Theoretical Framework for Computer Models of cooperative Dialogue, Acknowledging Multi-Agent Conflict", PhD Thesis, Open University, UK, July 1988.

[7] J.S. Rosenschien and M.R. Genesereth, "Deals Among Rational Agents", Proc. of the 9th Joint Conference on AI, Los Angeles, USA, pages 91-99, November 1985.

[8] P.E. Renaud, "Introduction To Client/Server Systems: A Practical Guide For Systems Professionals, 2nd Ed.", Wiley & Sons, ISBN 0-471-13333-7, June 1996.

[9] J. Stamos and G. Gifford, "Remote Evaluation", In ACM Transactions on Programming Languages and Systems, 12(4), pages 537-565, October 1990.

[10] J.R. Falcone, "A Programmable Interface Language for Heterogeneous systems", In ACM Transactions on Computer Systems, 5(4), pages 330-551, November 1987.

[11] K. Rothermel and R. Popescu-Zeletin, (Eds.), "Mobile Agents", Lecture Notes in Computer Science Series No 1219, Springer-Verlag, ISBN 3-540-62803-7, April 1997.

[12] L. Alex, G. Hayzelden and J. Bigham, "Software Agents in Communications Network Management: An Overview", Intelligent Systems Applications Group, UK, May 1999.

[13] F. Somers, "HYBRID: Intelligent Agents for Distributed ATM Network Management", Proc. of the IATA Workshop at ECAI'96, Budapest, Hungary, August 1996.

[14] S. Appleby and S. Steward, "Mobile software agents for control in telecommunications networks", In BT Technology Journal, 12(2), pages 104-113, April 1994.

[15] N. Minar, K.H. Kramer and P. Maes, "Cooperating Mobile Agents for Mapping Networks", Proc. of 1st Conference on Agent Based Computing, Hungary, May 1998.

[16] P. Dasgupta, N. Narasimhan, L.E. Moser and P.M. Melliar-Smith, "MAgNET: Mobile Agents for Networked Electronic Trading", University of California, 1999. Home page: http://alpha.ece.ucsb.edu/~pdg/research/papers/MAgNEThtml/MAgNET.html

[17] A. Sahuguet, "About Agents and Databases", CIS650, white paper, University of Pennsylvania, USA, May 1997.

[18] A. Hooda, A. Karmouch and S. Abu-Hakima, "Nomadic Support Using Agent-level Communication", Proc. of the 4th Symposium on Internetworking, Canada, July 1998.

[19] W.M. Farmer, J.D. Guttman and V. Swarup, "Security for Mobile Agents: Issues and Requirements", Proc. of the 19th National Information Systems Security Conference, pages 591-597, Baltimore, Md., October 1996.

[20] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla and G. Cybenko, "Agent Tcl: Targeting the Needs of Mobile Computers", IEEE Internet Computing, 4(1), pages58-67, August 1997.

[21] R.S. Gray, D. Kotz, G. Cybenko and D. Rus, "D'Agents: Security in a multiple-language, mobile-agent system" In Giovanni Vigna, ed., Mobile Agents and Security, Lecture Notes in Computer Science No 1419, pages 154-187, Springer 1998.

[22] H. Peine and T. Stolpmann "The Architecture of the ARA Platform for Mobile Agents", In [11]. ISBN 3-540-62803-7, Springer 1997.

[23] H. Peine, "An Introduction to Mobile Agent Programming and the Ara System", ZRI report, Dept. of Computer Science, Kaiserslautern U., Germany, January 1997.

[24] "Aglets Software Development kit", IBM Corporation. Home page: http://www.trl.ibm.co.jp/aglets/.

[25] G. Karjoth, D. Lange and M. Oshima, "A Security Model for Aglets", In IEEE Internet Computing, 1(4), pages 68-77, July-August 1997.

[26] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young and B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents", in [11], pages 86-97, Springer 1997.

[27] A.Castillo, M. Kawaguchi, N. Paciorek and D. Wong, "Concordia ™ as Enabling Technology for Cooperative Information Gathering", Proc. of the 31st Annual Hawaii International Conference on System Sciences (HICSS31), Kona, Hawaii, January 1998.

[28] J.E. White, "Mobile Agents", Technical report, General Magic, Inc., October 1996.

[29] G. Glass, "Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing", CTO ObjectSpace, 1999.

[30] G. Glass, "The ObjectSpace Voyager Universal ORB", CTO ObjectSpace, 1999.

[31] IKV++ GmbH, "Grasshopper: A Platform for Mobile Software Agents", Informations- und Kommunikations- technologie, Germany, 1999.

[32] Joint submission, "MASIF Specification", OMG TC Document orbos, October 1997.

[33] "Foundation for Intelligent Physical Agents".
Home page: http://www.fipa.org/

[34] C. Baumer, M. Breugst, S. Choy and T. Magendanz, "Grasshopper- A universal agent platform based on OMG MASIF and FIPA standards", Proc. of the 1st Workshop on Mobile Agents for Telecommunications Applications, Ottawa, Canada, October 1999.

[35] A. Karmouch and V. A. Pham, "A Mobile Agent-based Architecture in a Networking Environment", Report prepared for Nortel Technology, Ottawa, Canada, January 1999.

[36] V. Jayaraman, V.A. Pham and A. Karmouch, "Secure and Efficient Communication Infrastructure for a Mobile Agent System", white paper, Ottawa U., November 1998.

[37] B. Schulze, "Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture", in [11], Springer 1997.

[38] A. Karboubi, "Mobile Agent Management Framework", M.A.Sc.Thesis, University of Ottawa, Canada, April 2000.

[39] F. Hohl, "An approach to solve the problem of malicious hosts", Universität Stuttgart, Fakultät Informatik, Stuttgart, Germany, March 1997.

[40] G. Vigna, "Protecting Mobile Agents through Tracing", Proc. of the Second ECOOP Workshop on Mobile Object Systems, Finland, June 1997.

[41] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification (2nd Ed) (Java Series)". Addison-Wesley, ISBN 0201432943, April 1999.

[42] "Lightweight Directory Access Protocol".
Home page: http://www.umich.edu/~dirsvcs/ldap/.

[43] W. Yeong, T. Howes, S. Kille, "LDAP: RFC 1777", March 1995.
Home page: http://sunsite.auc.dk/RFC/rfc/rfc1777.html

[44] C. Apple, K. Rossen, "X.500 Implementations Catalog-96: RFC 2116", Network Working Group, April 1997.

[45] Netscape Communications Corporation, "Directory Server Documentation", 1999. Home page: http://devedge.netscape.com/docs/manuals/directory.html

[46] "Java Core Reflection". Home page: http://java.sun.com/products/jdk/1.2/docs/guide/reflection/spec/java-reflectionTOC.doc.html

[47] R. L. Rivest, A. Shamir and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", In Communications of the ACM, 21(2), pages 120-126, February 1978.

[48] National Institute of Standards and Technology, "Digital Signature Standard," NIST FIPS PUB 186, U.S. Department of Commerce, May 1994.

[49] National Institute of Standards and Technology, FIPS PUB 180-1, "Secure Hash Standard", U.S. Department of Commerce, May 1993.

[50] ITU-T Recommendation, "X.509: The Directory - Authentication Framework", 1988.

[51] Netscape Communications Corporation, "Introduction to SSL", 1998. Home page: http://developer.netscape.com/docs/manuals/security/sslin/index.htm

[52] A.V. Hoff, H. Partovi and T. Thai, "Open Software Description Format", August 1997. Home page: http://www.w3.org/TR/NOTE-OSD.html

[53] E.R. Harold, "XML Bible", IDG Books Worldwide, ISBN 0764532367, July 1999.

[54] M.J. Petrovsky, "Implementing CDF Channels (Hands-On Web Development)", Computing McGraw-Hill, ISBN 0070498873, February 1998.

[55] J.D. Moffett and M. Sloman, "The Representation of Policies as System Objects", Proc. of the Conference on Organizational Computer Systems, Atlanta, USA. In SIGOIS Bulletin, 12(2 & 3), pages 171-184, November 1991.

[56] D. Marriott and M. Sloman, "Management Policy Service for Distributed Systems", Proc. of the IEEE 3rd Int. Workshop on Services in Distributed and Networked Environments. Macao, June 1996.