Universitat Autònoma de Barcelona

Departament d'Enginyeria de la Informació i de les Comunicacions

EFFICIENT MOBILITY AND INTEROPERABILITY OF SOFTWARE AGENTS

SUBMITTED TO UNIVERSITAT AUTÒNOMA DE BARCELONA
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

by Jordi Cucurull Juan

Bellaterra, September 2008

Advisers:
Dr. Joan Borrell Viader
Dr. Guillermo Navarro Arribas

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bellaterra, September 2008

Dr. Joan Borrell Viader
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bellaterra, September 2008

Dr. Guillermo Navarro Arribas
(Adviser)

Abstract

Mobile agents are autonomous software entities that have the ability to stop and resume their execution in different network locations to accomplish a set of tasks. Despite their apparent simplicity, the fact of sharing a code in different places, in most cases heterogeneous, arises a set of issues which are far from have a simple solution. The proof is that after several years of efforts, a wide-scale deployment of mobile agents has not become a reality. In our opinion the main reasons which have hindered the adoption of mobile agents are: security, interoperability, and efficiency. Security may impose strong restrictions to the use of mobile agents. Nevertheless, enough research to satisfy the most common applications has been done in this field. Interoperability is absolutely indispensable to guarantee that different types of agents can run in different places and exchange information. And efficiency is a non functional requisite which favours the adoption of the technology.

The suitability of mobile agents for distributed and heterogeneous environments is unique. The work presented in this thesis is motivated by this fact and comprises four objectives to improve, in this order, the interoperability, efficiency, and security of mobile agents in the context of the IEEE-FIPA standards. The first objective is the design of a flexible agent mobility specification. The second objective is the proposal of interoperability mechanisms to move and execute agents in several locations supporting different agent middlewares, programming languages, and underlying architectures taking advantage of the previous mobility specification. The third objective is the proposal of methods to improve the efficiency of the agent mobility and interoperability. And finally, the last objective is the design of some protocols to deal with specific security issues of mobile agents.

Acknowledgements

In this preface I want to acknowledge several people for their support during my PhD studies. Since most of them are Catalan speakers I have written the rest of this section in Catalan. Nevertheless, before switching to it, I want to thank the people from the Vrije Universiteit (Benno, Michel, Reinier, Sander, Martijn, Thomas, Maik, Reza, Frances, Guido) their warm welcome during the three months I spent in Amsterdam.

La vida està formada per una sèrie d'etapes que, una vegada superades, solem analitzar, etiquetar i classificar en funció d'on ens hagin conduit. Els estudis de doctorat fàcilment es poden associar a una d'aquestes etapes que, a més de la feina, inclouen tota una sèrie de vivències personals. És per aquest motiu que faré extensius aquests agraïments, no tan sols a les persones que han tingut una relació directa amb el desenvolupament d'aquesta tesi, sinó també a les persones que han estat presents en aquesta etapa de la meva vida.

Així en primer lloc voldria agraïr als meus pares, així com a la meva família, el suport i l'ajuda que sempre m'han donat, especialment en moments difícils. De la mateixa manera vull agraïr als meus amics més propers (Raúl, Òscar i David) les estones que hem passat, tan en moments bons com en no tan bons. Agraïments també a l'Anna per la seva aparició del no res i per donar-me una mica de llum. També vull donar les gràcies als meus companys del departament pels dinars, estones divertides i moments tan agradables que m'han fet passar (Juan, Lino, Jorge, Joanet, Fernando, Cristina, Maria, Ramon, Guille, Sergi, Mari Carmen, Rosa Maria, Ian, Abraham, Carlos...). I una menció especial al meu darrer company de despatx, en Carles, amb qui he compartit de molt a prop el procés de redacció tan de la seva com de la meva tesi. I agraïr també les estones que hem passat als meus companys de les Jornades Doctorials.

Respecte a la tesi, vull donar les gràcies en primer lloc al meu supervisor, en Joan Borrell, per haver-me guiat en la seva escriptura i desenvolupament. També vull fer especial èmfasi amb en Guille, que a més de ser un gran company ha estat un excel·lent codirector. Vull donar les gràcies a en Ramon per les seves correccions. A en Sergi per les seves suggerències i per dirigir el grup. I a en Joan Ametller per iniciar-me en la mobilitat dels agents. I un agraiment també als alumnes de projectes que he tingut (Ferran, Víctor i Gerard) i que han col·laborat activament en la implementació d'alguns prototips relacionats amb la migració dels agents. Donar les gràcies, també, a en Jaume Pujol per haver-me vingut a buscar quan encara no havia acabat la carrera. I finalment, un agraïment a la resta de membres del departament per aportar, cadascú, el seu granet de sorra per a què tot tiri endavant.

Vull fer constar, a més a més, que aquest treball, a excepció del què es deriva de l'estada a la Vrije Universiteit, ha estat possible gràcies a la contribució econòmica que ha aportat el Departament d'Universitats, Recerca i Societat de la Informació de la Generalitat de Catalunya mitjançant els Fons Socials Europeus. Destacar també el finançament rebut dins del projecte Espanyol TSI2006-03481.

Acronyms

ACDS Agent Code Distribution Service

ACL Agent Communication Language

AID Agent Identifier

AM Agent Middleware

AMM Agent Mobility Manager

AMS Agent Management System

AP Agent Platform

ASIPMS AgentScape Inter-Platform Mobility Service

CAI Common Agent Interface

CGID Code Group Identifier

CID Code Identifier

CDN Content Delivery Network

DF Directory Facilitator

DHT Distributed Hash Table

EE Execution Environment

FrTP Fragmented Transfer Protocol

HAP Home Agent Platform

HCID Hash Code Identifier

JIPMS JADE Inter-Platform Mobility Service

IPMA Inter-Platform Mobility Architecture

IPMS Inter-Platform Mobility Service

MA Mobile Agent

MAS Mobile Agent System

MID Migration Identifier

MMP Main Migration Protocol

ODTP On-Demand Transfer Protocol

OSAAP One-Shot Agent Authentication Protocol

P2P Peer to Peer

PCTP Push Cache Transfer Protocol

PL Programming Language

PoP Point of Presence

PS Protocol Sequences

RESTTP REST Transfer Protocol

SR Security Revision

UA Underlying Architecture

VM Virtual Machine

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile Agents (MAs) are autonomous software entities that have the ability to stop and resume their execution in different network locations to accomplish a set of tasks. They live within environments called Agent Platforms (APs), which are composed of a software called Agent Middleware (AM), that define the boundaries of available locations and the basic functionality the agents have. This technology took its first steps with the publication of the Jim White's article [Whi96] in '96.

Despite the apparent simplicity behind MAs, a lot of research efforts have been spent along these years. The fact of sharing agent code in different locations, in most cases heterogeneous, arises a set of issues which are far from having a simple solution. The most common issues are related to the agent interoperability [MR00, MPD+02, PR02, GGK+02, OdGWB06, FGR07, FIP02a, OMG97] and security [KAG98, Yee99, JK00, Rot02, MB03, GMB+08] with quite a lot research associated. Nevertheless, after several years of efforts, a wide-scale deployment of MAs has not become a reality. This is the reason why, some time ago, several articles [Rot04, Gra04, Vig04] questioning the future of this technology arose. These articles coincide in that security is one of the main reasons that have hindered the adoption of MAs. But, other reasons appear, such as the lack of a killer application, the lack of a widespread infrastructure to host the agents which would not be created without promising applications, and so on.

In our opinion the main reasons which have prevented the adoption of MAs are security, interoperability, and efficiency. Security may impose strong restrictions to the

use of MAs. Nonetheless, enough research to satisfy the most common applications has been done in this field (Chapter 7). Interoperability [PR02] must be taken into account in highly distributed heterogeneous environments. Since MAs need to interoperate with elements and other agents present in these environments, a strong degree of interoperability is absolutely indispensable to guarantee the deployment of the technology and the new coming applications. As in security, different works exist in this area. Specially important are the standardisation efforts carried out by the OMG [OMG97] and the IEEE-FIPA [FIP02a] organisations. And, last but not least, the efficiency of MAs is another important issue to cope with. Theoretical technologies which once implemented do not perform efficiently, turn into non usable technologies without suitable applications for them. One of the criticisms to MAs is, precisely, a lack of efficiency [Vig04]. Although several works in this field have been done [Gav04, BR05], they are not integrated with interoperable solutions.

Currently, the deployment of MAs is limited, but new applications are constantly coming out. For example, Gavalas *et al.* [GGGO02] propose an approach to deal with the management of networks with changing conditions. Vimercati *et al.* [VFL06] propose the use of MAs for remote control and calibration of general purpose instrumentation, and remote measurement. Vieira-Marques *et al.* [VMRC+06] propose the secure integration of medical data using MAs. And, Marsá *et al.* [MMLCVN08] propose an approach to personalise the services users access in smart environments, such as home and offices, by using mobile personal agents.

1.1 Objectives

We are conviced that MAs are not an outdated technology. Their suitability for distributed and heterogeneous environments is unique. In our opinion their expansion will be at the right moment, when the advantages outweigh the disadvantages, and the surrounding technologies will be matured enough to support them. Our assumption is that by improving their interoperability, efficiency, and security, a wide-scale expansion is more probable.

The main objective of this thesis, which is motivated by this assumption, is to propose efficient mechanisms, within the context of the IEEE-FIPA standards, to guarantee the interoperability of heterogeneous MAs and AMs. Despite several works about the interoperability of agents exist, there are no mobile agent standards which guarantee a complete interoperability between all types of MAs and AMs. Standardisation organisations such as OMG and IEEE-FIPA, this last the most accepted nowadays, have devised several agent standards for AMs and for agent communication. Nevertheless, they do not include a robust specification for mobility and execution of agents in AMs that do not share the same profile. The main objective of the thesis has been splitted into four smaller objectives.

The first one is the design of an efficient, flexible, and extensible agent mobility specification to allow the migration of agents between different, and possibly heterogeneous, locations. This objective does not comprise the interoperability of the agent code in terms of execution, but it only comprises its transport.

The second objective is to devise methods to allow the execution of agents in several APs supporting different AMs, Programming Languages (PLs), and Underlying Architectures (UAs). The interoperability of agent communication, solved by IEEE-FIPA, and agent mobility, the previous objective, is not sufficient for a wide-scale mobile agent deployment. The achievement of them allows an agent to communicate with other agents and to visit any other location. Nevertheless, in case the visited locations have heterogeneous AMs or UAs the agent cannot be executed there.

Another objective is the proposal of methods to improve the efficiency of the agent mobility and interoperability. The efficiency of an agent migration is always constrained by the fact that all the agent components (agent code, data, and state) must be sent to the destination location. Therefore, different migration strategies for the mobility specification of the first objective, and agent code distribution methods can be proposed. Furthermore, the interoperability methods of the second objective must also be efficient.

As previously stated, the security of agents is also of outmost importance. Despite several research has been done in this area, another objective of the thesis is the contribution to the improvement of existing mobile agent security mechanisms. Furthermore, the security must be taken into account in the objectives previously detailed.

1.2 Structure

In the following paragraphs an outline of the chapters and appendices that compose this thesis is presented.

Chapter 2 introduces the mobile agent technology. First of all a definition of MAs is stated, then there is an introduction to agent basics, agent mobility, a discussion of the most frequent interoperability issues and extended solutions, and a summary of the most relevant agent security requirements and threats. More specific background is detailed in each chapter as appropriate.

Chapter 3 explains the design, implementation, and comparison of a new mobility model called Inter-Platform Mobility Architecture (IPMA), in the context of the IEEE-FIPA specifications, suitable for different kinds of AMs, and focused on improving the interoperability in the area of agent mobility. Furthermore, a set of migration transfer protocols, used to transfer the agent components, for the model devised are also explained. And finally, two implementations of the model and the protocols are shown.

Chapter 4 presents a set of tests to evaluate the most relevant performance differences between the migration transfer protocols proposed for IPMA: the Push Cache Transfer Protocol (PCTP), the On-Demand Transfer Protocol (ODTP), the Fragmented Transfer Protocol (FrTP), and the REST Transfer Protocol (RESTTP).

Chapter 5 proposes a global cache service to efficiently and securely deal with the distribution of agent code. An implementation of the service is also presented, and a set of agent migration performance tests demonstrate its benefits regarding the migrations without the service enabled.

Chapter 6 presents several mobile agent interoperability proposals to achieve the challenge of agents freely migrating in heterogeneous environments. Each of the included proposals increase a bit the final complexity of the system, but they improve the final solution. The chapter also includes a final comparison with other proposals existing in the literature.

Chapter 7 introduces several existing security mechanisms and two new methods to protect MAs. The first is a scheme to protect agent itineraries (list of locations that an agent visits) with loops. And the second is a protocol to authenticate agents and guarantee the integrity of their code in the mobility model proposed in Chapter 3.

Chapter 8 summarises the conclusions obtained from the objectives previously mentioned, and provides some future research directions on the topics dealt with.

Finally, Appendix A contains technical details about the mobility model of Chapter 3, and Appendix B contains two examples of an interface presented in Chapter 6.

1.3 Publications

The work along this thesis has produced several publications in conferences, books, and journals:

- J. Cucurull, J. Ametller, and J. Borrell. Protocol for the protection of mobile agent itineraries with loops (in Spanish). In Alberto Peinado Domínguez et al., editor, *1r Simposio sobre Seguridad Informática [SSI'2005]*, pages 61–68, Granada. Spain, September 2005. CEDI 2005, Thomson.

- J. Cucurull, J. Ametller, J.A. Ortega-Ruiz, S. Robles, and J. Borrell. Protecting mobile agent loops. In T. Magendanz, K. Ahmed, and I. Venieris, editors, *Mobility Aware Technologies and Applications*, volume 3744 of *Lecture Notes in Computer Science*, pages 74–83, Montreal, Canada, October 2005. MATA 2005, Springer.

- J. Ametller, J. Cucurull, R. Martí, G. Navarro, and S. Robles. Enabling mobile agents interoperability through fipa standards. In M. Klusch, M. Rovatsos, and T.R. Payne, editors, *Cooperative Information Agents X*, volume 4149 of *Lecture Notes in Artificial Intelligence*, pages 388–401, Edinburgh, UK, September 2006. CIA 2006, Springer Verlag.

- P. Vieira-Marques, S. Robles, J. Cucurull, R. Cruz-Correia, G. Navarro, and R. Martí. Secure integration of distributed medical data using mobile agents. *IEEE Intelligent Systems*, 21(6):47–54, 2006.

- J. Cucurull, J. Ametller, and R. Martí. Agent mobility. In F. L. Bellifemine, G. Caire, and D. Greenwood, editors, *Developing Multi-Agent Systems with JADE*, pages 115–130. Wiley, January 2006.

- J. Cucurull, R. Martí, S. Robles, J. Borrell, and G. Navarro. FIPA-based interoperable agent mobility. In *Multi-Agent Systems and Applications V*, volume 4696 of *LNAI*, pages 319–321, Leipzig, Germany, September 2007. Springer.

- J. Cucurull, B. J. Overeinder, M. A. Oey, J. Borrell, and F. M. T. Brazier. Abstract software migration architecture towards agent middleware interoperability. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 27–37, Wisla, Poland, October 2007.

- J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, B. J. Overeinder, and J. Borrell. Agent mobility architecture based on IEEE-FIPA standards. Submitted to Computer Communications.

- J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, and J. Borrell. Full mobile agent interoperability in an IEEE-FIPA context. Submitted to Journal of Systems and Software.

- J. Cucurull, G. Navarro-Arribas, R. Martí, S. Robles, and J. Borrell. An efficient and secure agent code distribution service. Submitted to Journal of Network and Computer Applications.

Chapter 2

Mobile Agents

This chapter introduces the mobile agent technology. Central part of it are the definition of mobile agents, the introduction of agent basics, the introduction to agent mobility, the discussion of the most frequent interoperability issues and extended solutions, and a summary of the most relevant agent security requirements and threats.

2.1 Definition

Mobile Agents (MAs) [Whi96] are a technology which has its origin on two different disciplines. On the one hand, the artificial intelligence community created the concept of intelligent agent [WJ95]. On the other hand, the distributed systems community, with a more pragmatic vision of MAs, focused on the exploitation of code mobility [FPV98].

A valid definition for MAs, regarding the two mentioned disciplines, is that they are intelligent software entities that have the ability to stop and resume their execution in different network locations to accomplish a set of tasks. They live within environments called agent platforms, that define the boundaries of available locations, and they are mobile, autonomous, reactive, proactive and social (see definitions in Table 2.1).

Although there is no application that cannot be conceived without the existence of MAs, they ease the implementation of applications which require:

- **Task delegation**: Due to the inherent autonomy of agents and MAs, they can be assigned with a set of tasks which the agent performs on behalf of its owner, e.g.,

Mobility	Agent ability of suspending its execution in a specific agent platform, and resume it in another agent platform, i.e., in another location. This process is usually called agent migration.
Autonomy	Each agent is driven according to a code specially developed to achieve one or more goals. The agent actions are completely decided according to this code without direct intervention of other parties.
Reactivity	Agents react to the environment changes in order to achieve their goals.
Proactivity	Agents change their environment and take several initiatives to achieve their goals.
Sociability	It is the ability of agents to interact with other agents. This is a key feature, since some agents only can perceive their environment through communication with other agents.

Table 2.1: Mobile Agent Properties.

in [VMRC+06] is described a medical application where the task of searching for patients information is delegated to an specific agent.

- **Asynchronous processing**: Mobile agent execution is not dependent on a continuous communication with the agent owner or the home agent platform. Therefore, the agent can freely move through different network locations while it carries out the assigned tasks. An example of this is nomadic computing [MMLVA06], where agents reside in mobile devices and migrate to other locations to perform tasks without consuming the scarce resources of the mobile device.

- **Dynamic environment adaptation**: Agents perceive environment changes and react by adapting their behaviour to them. An example applied to network management can be seen in [Sat03], where a MA is reused without modifications to manage various networks.

- **Flexible interfaces**: Since its ease of adaptability, agents can be used to interact with completely different interfaces, such as it is proposed in [VFL06]. Even, agents can be used as improvised adaptors between two kinds of interfaces.

- **Fault tolerance**: Because of the agents capacity to adapt to changing environments, MAs can easily deal with computer and network faults. They are specially

suitable for hostile environments, where the agent can decide to visit alternative locations in case of failure. An example of fault tolerance based on MAs can be seen in [KE06].

- **Parallelism**: The autonomous nature of MAs, the ability to migrate to different locations, and the capacity of interacting with other agents, make them suitable for parallel applications, where a coordinated group of several agents are used. An example, which uses them as a load balancing mechanism, can be seen in [TST+05].

- **Local data processing**: MAs can process data directly where it resides without having to move it from the original location. There are two kinds of applications which benefit from this feature. Firstly, *sea-of-data* applications where there is a large quantity of distributed information to process and the movement of it has an elevated cost [Gra03]. And, secondly, medical applications [VMRC+06] where moving data from its original location is not legal.

2.2 The Agent

A MA, from an architectural point of view, is an entity composed of a unique identifier and three main components: code, data, and state. Furthermore, an agent has a life cycle associated to its execution state. These components are maintained in Mobile Agent Systems (MASs) by Agent Platforms (APs). The software which implements APs, and therefore manages agents, is called Agent Middleware (AM).

2.2.1 Agent Identification

Each MA has an associated identifier that distinguishes it individually. This identifier is assigned when the agent is created, it should be immutable, and it is unique within the scope of the agent authority. The agent identification is usually related to the communication among agents. Although it is a well defined concept, each AM has its own approximation of it. In some cases the identifier is chosen by the agent developer, in other cases it is imposed by the AM, and some times an agent can have

more than one identifier. For example, the Aglets [LM98] middleware identifies every agent by using the AgletID, which is a unique identifier generated by the AM. On the other hand, the AgentScape [OB04] AM provides each of its agents with an identification called “agent handler”. This identification cannot be chosen by the agent developer, although more than one agent handler can be requested for each agent. And the JADE [BCPR08, BCG06] AM uses the Agent Identifier (AID) defined by IEEE-FIPA (Section 2.4.3), which in the case of JADE is composed of an agent name chosen by the agent owner in addition to the name of the home agent platform. Notice that in this last case, although the AP controls the names assigned to its agents, the agent owner is involved in the agent identification, therefore it has part of the responsibility for not creating agents with duplicate names.

2.2.2 Agent Components

The *agent code* is the core component of the agent and contains the agent’s main functionality. Several aspects of the agent code are important for the mobility and interoperability of agents. They are discussed in the next paragraphs.

The code is developed and compiled using a Programming Language (PL) and computer architecture supported by the hosting AMs. In some cases, for interoperability or efficiency reasons, even several versions of the code are provided with the agent (this is explained in more detail in Section 6.5), e.g., codes with the same functionality but developed with different PLs and/or compiled for different computer architectures.

The agent code is usually interpreted code, since it must be easily separable from its local agent platform for, later, being incorporated to a remote agent platform. This is the main reason why most of the MASs run over an interpreter or a virtual machine, e.g., the Java Runtime Environment. Interpreters even have been used with PLs such as C and C++, e.g., the MAS Mobile-C [CCP06] uses the C/C++ Ch [Sof] interpreter.

An important aspect related to the agent code is how it is managed and packed. Depending on the PL in which the code is written and the algorithm used in the agent migration, the code can be packed and sent in different ways. In object oriented PLs code is usually composed of many code snippets which represent each class and are packed in a single file. Usually this file is sent to remote platforms all at once, e.g., in

Java MASs the code is packed in JAR files. Nevertheless, there are on-demand migration algorithms (Section 3.4) where the code snippets are individually sent, in this case the code may not be packed in a single file. Therefore, the type of packaging used is highly dependent on the AM used and must be taken into account when interoperability between different MASs is pursued.

The *agent data* are the movable resources associated to the MA, i.e., all the information used and, maybe, produced by the agent during its life, which is moved along with it. In object oriented systems this is usually associated to the object instance. How this information is encoded is completely dependent on each AM, e.g., in Java MASs the Java Serialisation mechanism is typically used. In Section 6.5 a proposal for a common agent data encoding is presented.

The *agent state* is the information associated to the agent execution from a operating system point of view. It comprises the program counter, the heap, and so forth. Nonetheless, most of the code interpreters used in MASs do not support access to this information. In addition to that, the agent state stresses the interoperability issues, since the involved AMs must share the same code interpreters or virtual machines.

The solution commonly adopted when the capture of the agent state is not possible or convenient consists of replacing part of the agent state by the agent data. In this case, since no program counter is captured, the execution of the agent is always resumed from the first line of code. But the agent developer can use some tricks to save part of the execution state as agent data. An example is the use of switch control flow statements driven by a simple agent variable which is updated and saved in each agent execution. Therefore, the execution can be approximately resumed in a specific block of code. The use or not of the agent state leads to two different types of agent mobility, *strong* and *weak* mobility respectively, which are explained in Section 2.3. An example of two equivalent codes denoting these two cases is shown in Figure 2.1.

2.2.3 Agent Life Cycle

MAs, as any software process, have a finite period of life. During this period they are subject to different events such as their creation, suspension of their execution, migration, and so on. Any of these events may change the agent behaviour regarding their

Algorithm 1: Strong mobility.

```
begin
    Task A
    doMove();
    Task B
end
```

Algorithm 2: Weak mobility.

```
begin
    switch state do
        case 0
            Task A;
            state = 1;
            doMove();
            break;
        case 1
            Task B;
            break;
    end
end
```

Figure 2.1: Equivalent algorithms using strong and weak mobility.

environment. The possible behaviours the agent may adopt can be mapped as an agent state (do not confuse it with the agent state component previously explained). The agent life cycle is the set of possible states adopted by agents regarding the possible events that may be triggered to them.

Some AMs do not have the concept of agent life cycle, e.g., Tacoma [JLvR+02], Aglets [LM98], Tracy [BR05], and AgentScape [OB04]. In this case the most simple agent life cycle is applied. It consists of only two states, one applied to the agent when it is running and the other one when it is not, and two events, the agent creation and the agent death. Other AMs that take into account the concept of agent life cycle offer models with more states and events, e.g., JADE [BCPR08, BCG06], SeMoA [RJS01], Mobile-C [CCP06]. In this case, the agent life cycles can include states to indicate the agent is migrating, or it is suspended or waiting for some event among others.

2.2.4 The Mobile Agent System

Mobile Agent Systems (MASs) are the environments where agents reside, operate, and interact with each other. They provide the necessary infrastructure for the agents. MASs are typically composed of one or more APs that may be distributed through several hosts, and which are implemented by a software called AM. The AP typically offers

a set of basic services to agents, for example agent management, agent messaging, directory services, and so forth. The services offered are highly dependent on the AP implementation. Nevertheless, there is no general agreement in the specific details of MASs structure. Those which do not support agent mobility are called Multi Agent Systems. Although they share the MAS acronym, in this thesis only the Mobile Agent Systems (MASs) are considered, therefore there is no confusion about it. From now on we will consider MASs composed of a single AP residing in a specific *host*. The combination of an AP and a host will be called *location*. MAs are moved between locations.

Since the beginnings of the mobile agent technology [Whi96] several MASs have appeared. Describing all of them makes no sense since it is not required to understand all the chapters of this thesis. Therefore, only a selection of MASs are described in the specific chapters where they are required, e.g., see Section 6.2.

2.3 Agent Mobility

The main characteristic of MAs is the ability to migrate from one location to another. In this section there is an explanation of the mobility basics, the types of agent mobility, and the concept of agent itinerary.

2.3.1 Mobility Basics

Agent mobility is the ability of agents to suspend their execution, move their code, data, and state to another location, and there resume their execution. The set of actions involved in the movement of an agent is called *migration process*. The complexity of this process is variable and depends on the protocols and type of mobility chosen. The essential operations of a simple migration process are outlined in the following lines:

1. **Migration request**: An agent decides to migrate to another location or, in some cases, an agent requests another agent to migrate. Usually AM interfaces provide agents with methods to request the migration.

2. **Stop agent execution**: The execution of the migrating agent is stopped, therefore its life cycle is updated to the most appropriate state. From this step the agent can be unregistered from its local platform.

3. **Collect agent components**: The agent code, data, and state are collected and packed to be sent. As it is explained in Section 2.2.2 these components and their packaging are completely dependent on the MAS implementation.

4. **Transfer agent components**: The agent code, data, and state gathered in the previous step are sent to the remote location. How they are sent is also dependent on the MAS implementation, which in some cases can be compliant to some agent communication standards such as the IEEE-FIPA specifications described in Section 2.4.3.

5. **Agent rebuilding**: The agent is rebuilt and registered in the remote location by using the components previously transfered.

6. **Agent resumption**: The agent execution is resumed, therefore its life cycle is updated to the most appropriate state.

Although a simple migration process is not complex, real systems involve a large quantity of factors that can enormously increase its complexity, such as mobility protocols, or mobility interoperability. The analysis of these factors and their solution is one of the contributions of this thesis.

2.3.2 Types of Agent Mobility

There are two main types of agent mobility [FPV98]: *weak* and *strong* mobility. As it is illustrated in Figure 2.1, the type of mobility chosen dictates the agent code development style [CLZ00].

From a conceptual perspective, strong mobility is preferred. It allows agents to suspend their execution and, then, resume it exactly at the same point it was suspended. However, strong mobility is complex to implement since the execution state must be

captured and restored in the destination location, an operation which is not always supported by PLs and/or virtual machines, e.g., the Java Runtime Environment [CLZ00]. Nevertheless, there are some alternatives to implement strong mobility without having access to the execution state. They consist of the modification of the agent code, in the remote location, to make the next line to execute, the first line of code. This method is used in [Yee99] and described in [HY98]. Some variants of it are also proposed in [IKKW01, WHB01, CHB03]. Despite these solutions, there is another issue with the strong mobility. It is the high dependence on the underlying computer architectures or virtual machines, which makes difficult to achieve a minimal interoperability when heterogeneous systems are used. Some mobile agent platforms providing strong mobility are Telescript [Whi96] and D'Agents[GCK+02] (Section 6.2.2).

Weak mobility does not capture the execution state, then the code is always resumed from the beginning. This is not a major issue, because using the agent data, the programmer can drive the migration to a specific part of the code, e.g., using a switch control flow statement. This migration type is more difficult to manage by the agent developer, but it is the most flexible and portable alternative. Because of that, this type of mobility has been widely used for example in Aglets [LM98], Grasshopper [BBCM00], SeMoA [RJS01], and JADE [BCPR08, BCG06] (Section 6.2.1).

2.3.3 Agent Itineraries

Agent itineraries are the lists of locations that MAs visit during their life. The concept of itinerary was firstly introduced in the Concordia [WPT+97] AM. The concept is specially important when security is introduced to MAs (see Section 2.5 and Chapter 7 for a more detailed essay on agent security).

Two basic types of itineraries can be distinguished. On the one hand, there are *static itineraries*, which are decided when the agent is created. They comprise the set of ordered locations that the agent will visit during its life. And, on the other hand, there are *dynamic itineraries*, which are not initially preestablished and are decided during the agent life according to its necessities. In some cases this last kind of itineraries can be constrained to a subset of possible locations to visit.

There are also some intermediate proposals [SRM98, MB02, MB03] which are not

so restrictive as the static itineraries neither so open as dynamic itineraries. These proposals introduce new kinds of mobile agent transitions, i.e., the point when an agent decides the next location to visit from its itinerary. Instead of the single transition which makes the agent move from the present location to the next one, these new transition can make the agent to choose between visiting one from two selected locations, or visit both in an undetermined order among others. The set of possible types of transitions depends on the chosen scheme.

2.4 Agent Interoperability

For MAs to be deployed on Internet scale distributed systems, interoperability between different types of AM needs to be ensured. In the case of MAs this feature is specially important because agents must interact with other agents and different platforms.

2.4.1 Full Agent Interoperability

According to Pinsdorf *et al.* [PR02], two MASs are interoperable if a MA can interact and communicate with other agents (local or remote), and if the agents of one system can migrate to the other system, i.e., they can leave their system and resume their execution in the next interoperable system. This kind of interoperability will be called *full interoperability* from now on. Considering Ametller *et al.* [ACM+06], and, also, Pinsdorf *et al.* [PR02], several areas can be inferred to cope with this *full interoperability*:

- **Programming language and underlying architecture**: This area is not limited to MAs, but to any application. Since different PLs, operating systems, microprocessors, and Underlying Architectures (UAs) exist, it is impossible to have a unique executable code format. Some of the solutions to this problem exploit the use of interpreted PLs such as *Perl*, *Python*, or *Java*, where a virtual machine abstracts the code from the underlying hardware and operating systems.

- **Middleware**: Since agents run over AMs, there are important constraints about the Application Programming Interfaces (APIs), the agent management, and the agent life cycle model. Several solutions in the literature propose the use of

adaptation layers to provide an uniform set of middleware properties to all the agents [MPD+02], the use of wrapper mechanisms to provide voluntary interoperability [PR02], and the use of a high level approach based on the concept of the universal agent [FGR07].

- **Communication**: Different AMs usually implement different communication methods for their agents. This includes the message structures used and the message delivery methods. This problem can be solved by using well-known agent standards like the ones defined by IEEE-FIPA (Section 2.4.3).

- **Mobility**: Similarly to the communication area, migrating an agent implies agreement with the set of messages exchanged and with the methods of delivery used. In addition to that, the AM must agree with the steps of the migration process and the information exchanged in each message.

A *security* area could also be included, although it is not essential for reaching a *full interoperability* based on the previous definition.

2.4.2 Agent Standards

A number of organisations have initiated the development of agent standards, at platform and communication areas, in an attempt to deal with the problem of incompatibility and interoperability. The standards have had different degrees of success, although none of them tackle the four areas previously described.

The first organisation to deal with agent standardisation was the Object Management Group (OMG). OMG wrote a specification document called *Mobile Agent System Interoperability Facilities (MASIF)* [OMG97], which states a set of common interfaces (MAFFinder, a naming service, and MAFAgentSystem, for management tasks) and definitions based on the CORBA IDL specifications. It provides possible AM and mobility interoperability, although this cannot be achieved without the collaboration of the involved MASs developers. It is not intended to deal with agent mobility between different kinds of AM, and neither agent standard interface nor agent communication mechanisms are defined. Concerning security, MASIF simply addresses existing CORBA

security to fit within the mobile agents middleware as better as possible. Currently, MASIF has no activity and cannot be considered as an option in nowadays AMs. This standard was used in several AMs, such as Aglets [LM98], Grasshopper [BBCM00], SMART [WHN+01] or SOMA [BCS01b] (Section 6.2.1).

There is a second organisation dealing with agent standardisation, the *IEEE Foundation for Intelligent Physical Agents (IEEE-FIPA)*, which is focused on the management and communication of intelligent agents. The specifications standardised by IEEE-FIPA define the basic components of an agent platform, an agent identification scheme, a complete communication infrastructure, and several agent management services. IEEE-FIPA is a dynamic organisation, its members (universities, companies, research institutes, and so forth) are always actively proposing specifications for new areas of interest. Therefore, the IEEE-FIPA are a set of up-to-date standards that must be considered when implementing present AMs. IEEE-FIPA specifications are used in more recent AMs regarding MASIF, such as JADE [BCG06] or Mobile-C [CCP06] (Section 6.2.1).

2.4.3 IEEE-FIPA

The IEEE-FIPA organisation has standardised a set of specifications to guarantee agent interoperability in the areas of agent middleware and agent communication. The first specification is the FIPA Abstract Architecture [FIP02a] which describes the basic aspects of a multi agent system (take into account that IEEE-FIPA does not initially consider MAs). Nevertheless, the core of the IEEE-FIPA specifications is the FIPA Agent Management Specification[FIP04], which defines the agent management reference model, together with the FIPA ACL Message Structure Specification [FIP02e].

The Agent Management Reference Model

The agent management reference model is the normative framework where FIPA agents are executed. An agent platform compliant with IEEE-FIPA and, therefore, with this model, is composed of agents, an Agent Management System (AMS), a Message Transport System (MTS), and, optionally, a Directory Facilitator (DF). This is depicted in Figure 2.2. The internals of these components are not subject to standardisation within

IEEE-FIPA, giving developers a lot of independence to make their own implementations.

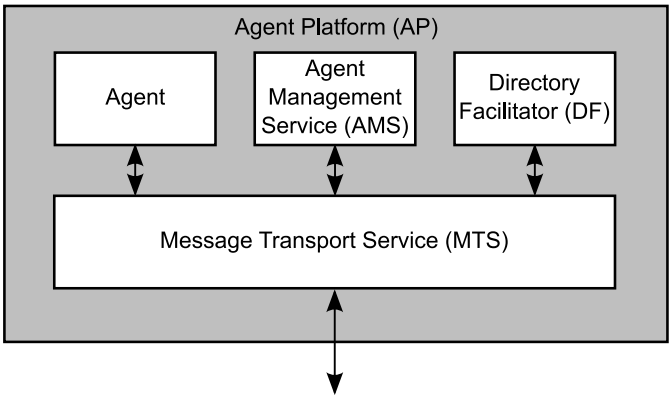

Figure 2.2: IEEE-FIPA Agent Management Reference Model.

The IEEE-FIPA agent is a computation process which is identified by an Agent Identifier (AID), and that communicates with other agents by using the Agent Communication Language [FIP02e] (ACL). The AID includes the agent name, which is mandatory, a set of transport addresses to reach the agent, and a set of name resolution service addresses. The agent name is an immutable globally unique identifier. No specific format is enforced, although it is proposed the use of a simple agent name concatenated with the home agent platform address (the platform where the agent has been initially created). This format is used in the JADE [BCPR08, BCG06] agent platform. The transport addresses are the physical addresses where the agent can be reached. One or more addresses can be provided, and they are encoded according to the URL format [BLFMa]. Finally, the name resolution service addresses are a set of agent AIDs which offer the name resolution service (typically implemented by the AMS). This is a service that returns an agent description for the requested AID supposing it would be registered there.

The Agent Management System (AMS) is a mandatory component which maintains a directory of agents with their corresponding AIDs, i.e., it is a white pages service. Each agent must be registered to this component in order to get a valid AID. Furthermore, the AMS, on behalf of the agent platform, maintains its agent's life cycles. IEEE-FIPA defines a specific agent life cycle in [FIP04]. It has six different states (*active*,

initiated, *waiting*, *suspended*, *transit*, and *unknown*) and ten possible transitions (*create*, *invoke*, *destroy*, *quit*, *suspend*, *resume*, *wait*, *wake up*, *move*, and *execute*), which are used, among other things, to manage the delivery of messages to the agent. Finally, only a single AMS exists in each agent platform.

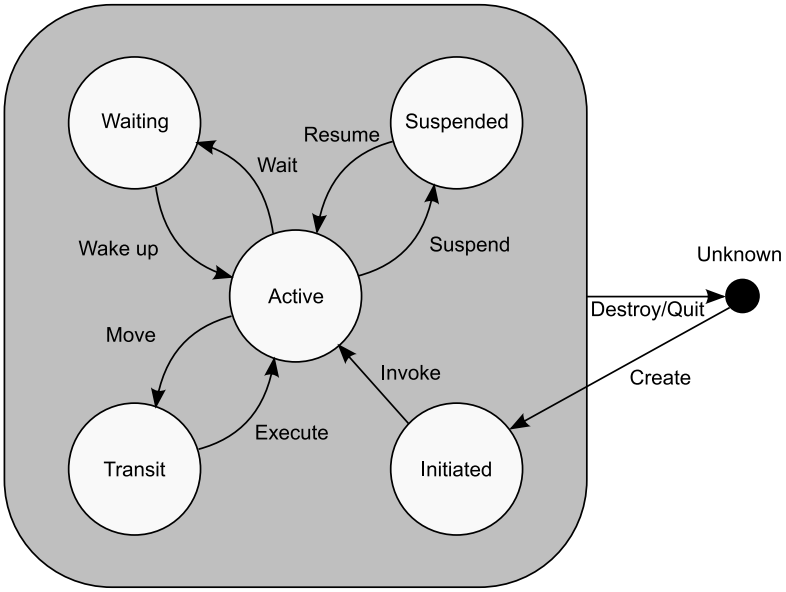

Figure 2.3: Agent Life Cycle.

The Message Transport Service [FIP02h] (MTS) is another mandatory component which provides local and remote agent communication. The service is offered through the Agent Communication Channel (ACC), which is integrated within the MTS and implements one or more Message Transport Protocols (MTP). An MTP is a physical protocol used to transfer the agent message data, e.g., the MTP-HTTP [FIP02f], which is based on the transfer of agent messages over the application network protocol HTTP, or the MTP-IIOP [FIP02g], which is based on the transfer of agent messages over an OMG IDL [OMG99] structure. According to [FIP02h] an agent can send a message to another agent through the local and remote ACC, directly to the remote ACC or directly to the other agent (by using proprietary mechanisms). In Figure 2.4 there is shown the first case which is the most common.

The Directory Facilitator (DF) is an optional component that provides a directory of services offered by agents. It is a yellow pages service. More than one DF can exist in

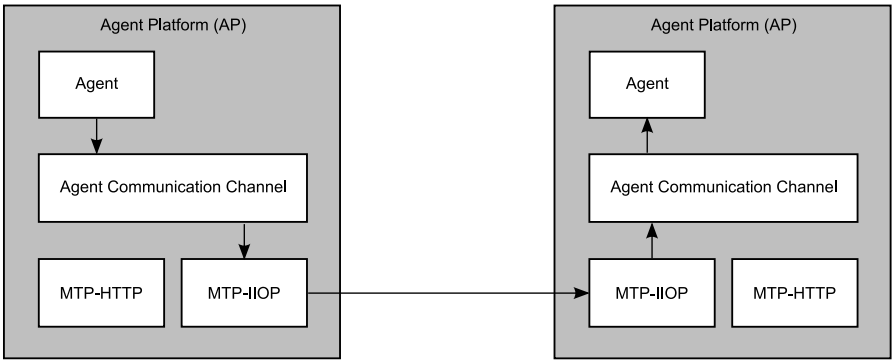

Figure 2.4: Agent Message Transfer.

each agent platform.

The Agent Communication Language

The communication of agents is not only limited to the transport of messages, but to the content and encoding of them. IEEE-FIPA has done an important effort in the definition of a standard Agent Communication Language (ACL), specified in the FIPA ACL Message Structure Specification [FIP02e].

The FIPA ACL message is composed of a set of one or more possible parameters which are listed on Table 2.2. The only mandatory parameter is the `performative`, which denotes the type of communicative act (defined in [FIP02i]) associated to the message. Usually, messages will also contain the `sender`, `receiver`, and `content` parameters. In addition to the parameters listed on Table 2.2, user-defined parameters can also be included in the message. Their name must always begin with the prefatory string "X-". The ACL message can be represented using different encodings, e.g., IEEE-FIPA defines specifications for encodings based on a Bit-Efficient representation [FIP02b], a String representation [FIP02c], and a XML representation [FIP02d]. Furthermore, the ACL message is encapsulated within a structure called envelope, which is also encoded using a specific representation.

The type of content included in an ACL message is described by the `language`, `encoding`, and `ontology` parameters. IEEE-FIPA does not enforce any specific

Parameter	Category
`performative`	Type of communicative acts
`sender`	Participant in communication
`receiver`	Participant in communication
`reply-to`	Participant in communication
`content`	Content of message
`language`	Description of content
`encoding`	Description of content
`ontology`	Description of content
`protocol`	Control of conversation
`conversation-id`	Control of conversation
`reply-with`	Control of conversation
`in-reply-to`	Control of conversation
`reply-by`	Control of conversation

Table 2.2: FIPA ACL Message Parameters.

type of content, since it is completely dependent on the agents involved. Nevertheless, a specific syntax and its associated semantics, based on the s-expression syntax, are suggested as a content language for the ACL messages. It is the Semantic Language (SL) [FIP02l] content language. Three cumulative subsets of the FIPA SL are defined. The FIPA SL0, the minimal subset, the FIPA SL1, with support for propositional expressions, and the FIPA SL2, with support for first order predicate and modal logic, although restricted to ensure that it must be decidable. The FIPA SL0 is used to encode the content of messages exchanged with the AMS according to the Agent Management Ontology specified in [FIP04]. FIPA SL0 is also extensively used in the Inter-Platform Mobility Architecture (IPMA), proposed in Chapter 3. This subset of the FIPA SL provides agents with support for requesting actions and state predicates.

Usually, ACL messages are involved in agent conversations. These conversations can be expressed in terms of interaction protocols, i.e., a protocol that defines exactly which ACL messages must be exchanged to carry out a specific action. Although there are several parameters related to the control of agent conversations, three are specific for the interaction protocols. The `protocol` parameter, which specifies the interaction protocol name in which the message is involved. The `conversation-id` parameter, which contains a non-null value assigned by the initiator entity of the protocol and which

is present in all the messages involved in the protocol. And the `reply-by` parameter, which indicates the latest time by which the sending agent would like to receive the next message of the protocol flow. IEEE-FIPA has released several specifications for different interaction protocols, e.g., the FIPA Request Interaction Protocol Specification [FIP02k] and the FIPA Propose Interaction Protocol Specification [FIP02j]. These two interaction protocols are represented using the Agent UML [BMO01, OPB01] notation in Figures 2.5 and 2.6 respectively.

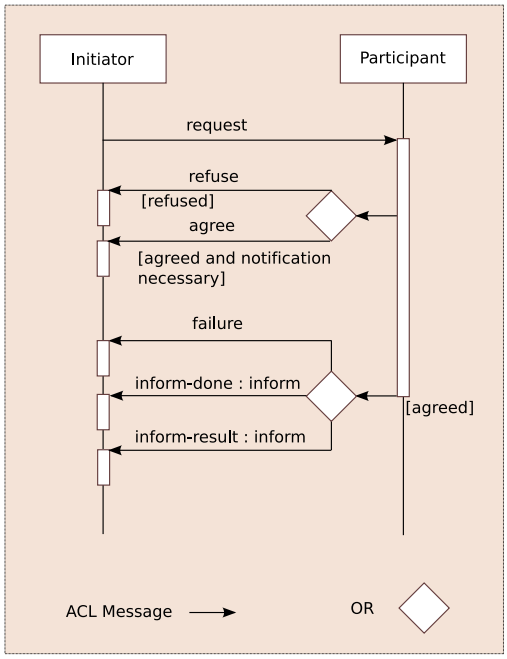

Figure 2.5: FIPA Request Interaction Protocol.

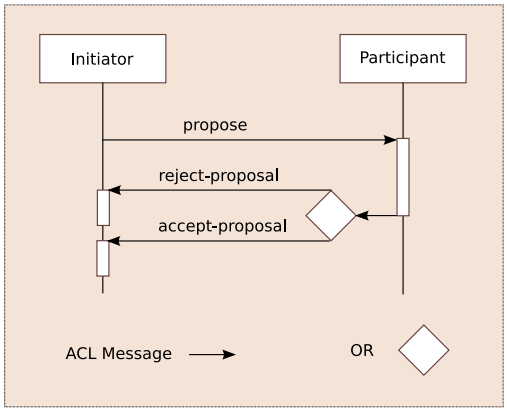

Figure 2.6: FIPA Propose Interaction Protocol.

2.5 Agent Security

The security of MAs is one of the topics with more research in the area. Despite its complexity, at the end, the security requirements desired for MAs are similar to the ones desired for any computer system. In general some security mechanisms are expected to prevent or, at least, detect system abuses. The most common security requirements are listed in the following lines:

- **Confidentiality**: Data carried, sent, and used by agents must be kept confidential regarding the other agents. Unauthorised access to this data may compromise the agent owner, APs, and other agents. Furthermore, it is also desirable to keep agent communications secret. Sometimes, even the message flow must be protected, since it can hint third parties about some agent operations.

- **Authenticity and integrity**: This requirement guarantees that agent data and code really belong to the entity which claims for their ownership, and that they are not manipulated. It is important to prevent the execution of hacked agents. It is also desirable to protect the agent results along their itinerary, preventing forgery.

- **Access control**: Only authorised agents must get access to a specific AP. Furthermore, a specific degree of privileges, with restrictions to get access to specific resources, must be assigned to agents. It is also usual to define which agents can use which resources. As can be seen in the literature [NA06], the MA access control is complex because of the distributed nature of agents, and because of the huge number of different authorities that can represent an agent.

- **Accountability**: This requirement implies the registration of all the actions carried out by agents, APs, and other entities participating of the system. Therefore, they have to legally account for their actions.

- **Availability**: APs must guarantee the availability of their services, despite the errors or denial of service attacks that may happen. Furthermore, fault tolerance systems should be present to detect and recover from possible AP failures and agent loses [SBS00, LCW04].

- **Anonymity**: In several cases, such as electronic business, the anonymity of the agent owner must be preserved. Nonetheless, this requirement must be balanced with the accountability requirement. Otherwise, agents would not take responsibility of their actions.

In addition to the security requirements, a set of usual mobile agent security threats are detailed in the following paragraphs. The threats listed are not exclusive of the mobile agent technology, but can be applied to it.

- *Unauthorised information disclosure*: The information from which agents are composed (code, data, and state) can be obtained by unauthorised parties. This also affects the information exchanged between two agents and information owned by APs.

- *Information manipulation*: The information related to agents can be manipulated by unauthorised parties. The consequences of this manipulation depend on the information modified: agent code modification changes the agent behaviour; agent data modification changes the agent results and behaviour; agent state modification changes the agent behaviour; agent itinerary modification changes the next locations to visit; and agent communications modification manipulates the agent interaction.

- *Denial of service*: Agents, APs, or third parties can attempt to make unavailable the services offered by other agents and APs.

- *Unauthorised access*: Agents may intend to gain privileges to migrate to locations or access resources which are forbidden to them.

- *Impersonation*: Agents and APs that fraudulently adopt the identity of another agent or AP. The aim of this attack is gaining privileges of other parties or doing things on behalf of them.

- *Repudiation*: Agents that deny something they have really done. This is important to prevent, otherwise agents could commit fraud and deny its responsibility.

- *Copy and re-execution*: Agents which are copied and executed in non authorised locations according to the agent itinerary. This is also applicable to the execution of an agent to an AP present in its itinerary.

2.6 Summary

In this chapter the mobile agent technology has been introduced. MAs have been defined and their properties have been discussed. Then, several important aspects for

the comprehension of the next chapters have been dealt. Firstly the agent internals have been presented, including their identification, the components from which they are composed, their life cycle, and the environments where they reside. Secondly, the mobility of agents has been dealt, explaining the basic steps needed to move an agent, the existing types of agent mobility, and the concept of agent itineraries. Thirdly, the interoperability of agents has been discussed. In this section the different areas which affect the interoperability of agents have been shown, some agent standards have been presented, and the IEEE-FIPA specifications have been detailed from the perspective required for the rest of the thesis. And, finally, the mobile agent security requirements and possible threats have also been presented.

Chapter 3

Inter-Platform Mobility Architecture

This chapter presents the design, implementation, and comparison of a new mobility model suitable for different kinds of agent middleware and focused on improving the interoperability in the area of agent mobility.

3.1 Introduction

Agent mobility, as stated in the previous chapter, is the ability of agents to suspend their execution, move their code, data, and state to another location, and there resume their execution. Although the concept is simple, the design of a mobility model suitable for different kinds of Agent Middleware (AM) and focused on achieving *full agent interoperability* (Section 2.4.1) is not an easy task.

In the next sections a flexible agent migration model, strongly based on well-known agent standards, is presented. Its main goal is dealing with interoperability in the area of agent mobility. The result is a flexible agent migration architecture, called Inter-Platform Mobility Architecture (IPMA), based on the IEEE-FIPA specifications and, therefore, conceived to be implemented at the application level of any AM with any Programming Language (PL).

The flexibility of IPMA is a consequence of its multi-protocol design. As it is explained in Section 3.3, some tasks of the migration process can be implemented by negotiable protocols. Therefore each migration can be different from the others. One

of the most relevant tasks is the transfer of the agent code, data, and state. The way these parts of the agent are transfered depends on the migration strategy [BR05] used in the migration process. Several migration strategies are presented in Section 3.4 as interchangeable protocols for IPMA: the Push Cache Transfer Protocol (PCTP), the On-Demand Transfer Protocol (ODTP), the Fragmented Transfer Protocol (FrTP), and the REST Transfer Protocol (RESTTP).

The contribution reported in this chapter has been implemented, in order to validate it, in the JADE AM (Section 3.6), and the AgentScape AM (Section 3.7), as the JADE Inter-Platform Mobility Service (JIPMS) [JIPa], and the AgentScape Inter-Platform Mobility Service (ASIPMS) [COO+07] respectively. This work in agent mobility was preceded by a first inter-platform mobility service prototype, also implemented in the JADE AM [CAM06], to evaluate the viability of using the mentioned IEEE-FIPA standards in agent migration [ARB03].

3.2 Related work

In this section several works related to the interoperability in the area of agent mobility are shown. Each one is focused on a set of specific goals.

The first contribution is the *Mobile Agent System Interoperability Facilities (MASIF)* [OMG97] specification created by the Object Management Group (OMG). As previously explained in Section 2.4.2, MASIF defines a set of common interfaces and definitions based on the CORBA IDL for mobile AMs. They merely provide a set of low level methods which developers implement in their AMs. There are no migration strategies imposed by the specification, although they may be limited to the possibilities offered by the combination of the methods provided. Nevertheless MASIF is not intended to deal with agent mobility between different kinds of AM, where different agent profiles are supported. Nowadays, MASIF can be considered outdated, since it is no longer adopted by new AMs.

Another contribution is the *FIPA Agent Management Support for Mobility Specification* [FIP00] created by the IEEE-FIPA organisation. The specification proposes two

migration protocols, some changes to the standard FIPA agent life cycle, and some additions to the FIPA Agent Management ontology. Nonetheless, this mobility proposal is only intended to be an application level wrapper for the native mobility mechanisms present in existing Mobile Agent Systems (MASs), i.e., it provides a set of high level tools to coordinate the migration of agents between two platforms. Furthermore, security concerns are not addressed by the specification. Finally, since it did not have sufficient acceptance and different independent implementations, it consequently did not get the classification of a standard. Nevertheless, the changes to the standard FIPA agent life cycle were finally integrated in [FIP04].

Out of the main standardisation initiatives, there are two other works which have interoperability as part of their goals: the Kalong mobility model [BR05] and the Agent Operating System (AOS) [vNOT+07]. *Kalong* is a mobility model focused on the achievement of efficient agent migrations. It is based on the Simple Agent Transmission Protocol (SATP), which defines a set of binary messages to support all the common migration operations (transfer of code snippets, transfer of data, commands to load code, and so forth). It has the advantage that, similarly to MASIF, the combination of these operations end up into different migration strategies (sending all the code at once or only the parts needed among others). Their authors refer to the implementation of Kalong as a virtual machine for agent migration (for its ability to execute the mentioned operations), and as a software component. Unlike the previous contributions, Kalong was designed to be used with the Tracy AMs, although it can also be used in other AMs as its own authors demonstrate in [PBK05].

Agent Operating System (AOS) [vNOT+07] is a specification which defines a layer between local operating systems and high level AMs. This layer guarantees interoperability in the areas of communication and mobility with other AMs using the same model. AOS is focused on security, and supports secure communication, secure agent storage and secure migration. It is used in the AgentScape [OB04] and the Mansion [vNBT04] AMs.

There are two main issues with the presented standards or specifications. Firstly, MASIF and IEEE-FIPA do not support a full mobility model, i.e., they are dependent on elements which are not comprised within the standard or the specification. Secondly,

Kalong and AOS, despite being full mobility models, are not based on major agent standards. These reasons have taken us to propose a new specification.

3.3 Inter-Platform Mobility Architecture

In this section a new inter-platform mobility architecture based on a full mobility model that uses well-known agent standards is presented.

3.3.1 Mobility Model

The mobility model described in this section is subject to the following list of requirements:

- *Full agent mobility model*: the model described must be complete, without depending on middleware native migration services or other non standard facilities.
- *Based on well-known agent standards*: the model must not be isolated from the existing agent technologies. Thus, the IEEE-FIPA specifications are chosen because they are the most widely used agent standards in current AMs.
- *Support for different migration strategies*: this must be an open model not limited to a set of fixed migration strategies. A mechanism supporting multiple, eligible, and negotiable migration strategies should be used.
- *Application-level oriented*: to ease the integration into existing AMs an application-level oriented migration model is required. Furthermore, this philosophy conforms to the IEEE-FIPA specifications.
- *Mobility type independence*: the type of mobility, i.e., weak or strong, depends on the availability of the agent state. The mobility model must be independent of this. It must provide appropriate tools to support the two cases, such as agent profiles and the ability to transfer the agent state in case of necessity.

- *Agent middleware independence*: the suitability of the model for any AM is an essential requirement to provide interoperability in the mobility area. This is the reason why the model presented is described in IEEE-FIPA terms.

- *Extensible for future requirements*: the agent technology is in constant evolution, therefore the mobility model cannot be closed to future requirements.

Taking into account the requirements listed above, an abstract migration model has been devised. In Figure 3.1 the general idea of the migration model is depicted. The model is designed to be implemented on top of any existing AM compliant with the IEEE-FIPA specifications. It is based on a regular agent called Agent Mobility Manager (AMM) which exchanges Agent Communication Language (ACL) Messages [FIP02e] with remote AMM agents to carry out the agent migrations. The use of ACL messages to migrate agents was firstly proposed in [ARB03].

Other implementations of the model without using a regular agent, such as the AMM, were possible, but they would not totally conform to the IEEE-FIPA specifications, since IEEE-FIPA does not consider other possible interactions than agent to agent. Furthermore, the AMM agent, as a mobility manager, can be in charge of other future mobility related tasks, such as agent tracking and message forwarding [CFL+02], local agent resource repository maintenance, and so on.

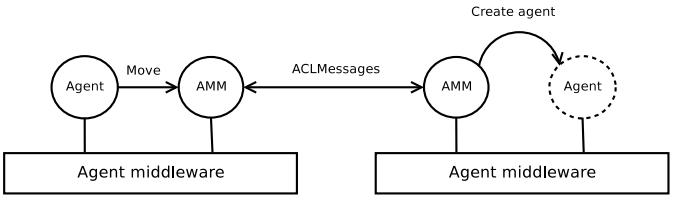

Figure 3.1: Migration model.

The general migration process supported by the model, from a high-level point of view, is depicted in Figure 3.2, and includes the following steps:

1. The agent contacts the local AMM and requests to migrate.

2. The local AMM suspends the agent execution, and starts the ACL message exchange with the remote AMM to agree with the next subprocesses.

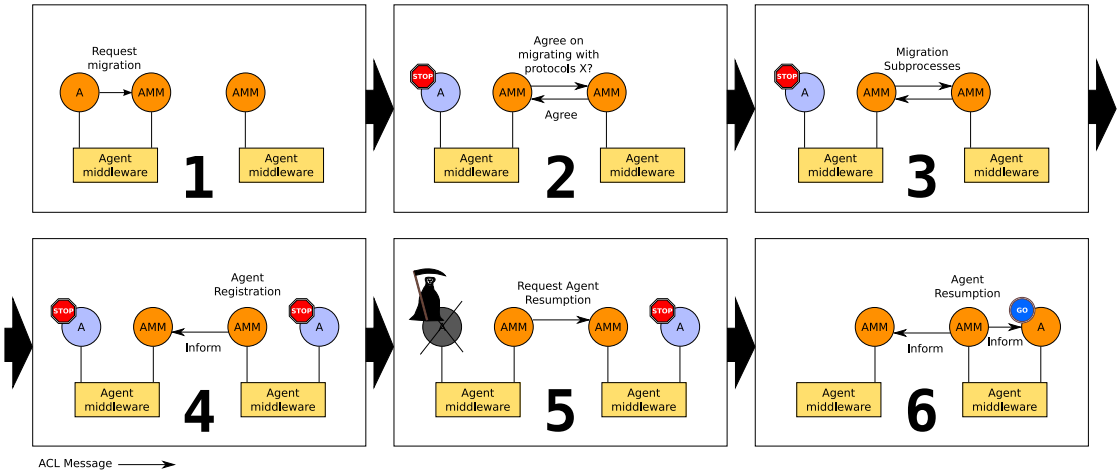

Figure 3.2: General migration process.

3. The agreed subprocesses are carried out:
   - optional agent authentication.
   - agent transfer (including agent code+data+state).
   - optional operations.
4. The remote AMM creates and registers the new agent using the code+data+state.
5. The local AMM shuts down the local agent and requests the remote AMM to start the agent.
6. The remote AMM starts the new remote agent and informs about the success of the process.

The migration model proposed is conceived to be implemented at the application level, minimising or avoiding the middleware internal modifications. Thus the migration is also initiated from this level. This is the reason why the migration of an agent is requested by sending a specific ACL message to the AMM agent (step 1). Therefore, the migration of an agent can be initiated by the agent itself or by another agent. Nevertheless, complementary methods can also be supported.

The rest of the process (steps 2–6) is carried out by the two involved AMMs following the basic operations specified in the architecture presented in Section 3.3.2. As

the whole mobility model, the architecture is based on the exchange of ACL messages, although some specific parts could take advantage of other technologies.

The process concludes with the migrating agent in the destination platform. This agent is an indistinguishable copy of the one that was present in the source platform, since the agent name is not modified by the migration process. The name of an agent cannot be changed, as stated in the FIPA Abstract Architecture Specification [FIP02a].

Finally, notice that during almost all the process the local agent has been maintained alive, although in a frozen state. The reason is that this mobility model is based on several steps which may involve several negotiations that can drive the migration to fail. In this case it is better to have the local agent ready to be resumed. The only expense is that, at then end, a last step to kill the local agent and resume the execution of the remote one is required. This step guarantees that only one copy of the same agent may be running at a time.

3.3.2 Mobility Architecture

The migration model proposed in the previous section is mapped to a specification called Inter-Platform Mobility Architecture (IPMA). The following paragraphs explain the basic aspects of it, the agent profile and code management, the service registration, the Main Migration Protocol (MMP), and the Protocol Sequences (PS).

Agent profile and code management

The proposed migration model claims its suitability for any AM. Nevertheless, this does not mean that all agents can migrate to any AM, since there may be differences in the elements comprised by the other interoperability areas (Section 2.4.1). This is why exists the concept of *agent profile*.

The agent profile is a table with compatibility information about the migrating agent and its code. According to this information a remote location decides to accept or refuse an incoming agent. Included information in the IPMA agent profile refers to the MAS, PL, and operating system where the agent should be executed (Table 3.1).

An agent, as proposed in Chapter 6, can extend its compatibility by providing several

codes suitable for different profiles. In this case, each agent profile table is associated to one of this codes by means of Code Identifiers (CIDs). Furthermore, the order in which agent profiles are included indicates the preferred order of code usage, since sometimes more than one code can be chosen in a specific Agent Platform (AP).

The agent code is identified by the combination of several elements, the Code Group Identifier (CGID), the Code Identifier (CID), the Security Revision (SR), and the Hash Code Identifier (HCID). An example of it is illustrated in Figure 3.3.

Group Code Identifier (CGID): *1f332e209d0dc354*		
Code Identifier (CID)	**Security Revision (SR)**	**Hash Code Identifier (HCID)**
22d1ccfe8a	*3*	*ebd6c336cd3cebf83f5410a180112d41*
cb3acb26ce	*1*	*f368f50481a99b0b44b083d24e5b03d6*
27de3d6279	*2*	*225ea33d20e0185d1f4a5aa271e482a4*
56a7c9daa8	*1*	*2a51345179ead3700c8b2fecd150e222*
27de3d6279	*1*	*aff05f9767ce0dbb4e5bed7ae67518fe*

Figure 3.3: Agent Code Identification.

- **Code Group Identifier (CGID):** Random value which uniquely identifies the group of all codes associated to an agent (including all revisions, and different PL codes compiled for different underlying architectures). It can help to add new agent codes for a specific agent once it has been started, since the new code can be bound to the existing agent.

- **Code Identifier (CID):** Random value which uniquely identifies an agent code with a specific functionality, developed in a specific PL, and compiled for a specific underlying architecture. The CID links an agent code with the specific profile exchanged during the agent migration. Code security revisions do not alter the CID.

- **Security Revision (SR):** Integer value which indicates the revision of the code in terms of security bugfixes. It is used to decide which agent code with the same CID is more recent.

- **Hash Code Identifier (HCID):** Unique value created from the XOR of a MD5 cryptographic hash function applied to each code snippet. An agent code is usually composed of a package (ZIP file or Jar file among others) that contains several smaller code snippets, e.g. classes. The HCID is used to check the integrity of this agent code, in addition to be used by AP embedded caching mechanisms. This identifier is resilient to agent code repackaging, which is carried out by some migration transfer protocols (Section 3.4), since it is calculated from the smallest code snippets.

Finally, another important aspect to take into account is the way the agent data and state are encoded. Different AMs may support different incompatible encodings. For example, the most used data encoding algorithm in Java AMs is the Java serialisation mechanism, but in fact several other methods are possible (see Section 6.5.1 for a list of them). IPMA enforces the use of only one agent data and one agent state encoding methods for each agent. Otherwise, the agent data and state could not be shared from one location to another (see Section 6.5 for a discussion about it).

Service registration

IPMA is designed as a service offered by the AMM agent. Such as other IEEE-FIPA services, it is recommended to be registered in the Directory Facilitator (DF) [FIP04] of the AP. DF is a yellow pages service optionally present in IEEE-FIPA compliant AMs. There are two reasons for publishing the migration service in a yellow pages service:

- Easy localisation of remote destinations that support IPMA. Possible locations can be searched in two ways. In the first way agents request remote DF agents belonging to possible destinations to check if they support IPMA. In the second way agents only request their local DF agent. Therefore it is assumed that it is federated with other DFs and it can obtain the services registered there.

- Assessment of local and remote locations support for the same migration architecture protocols and agent profiles. The local migrating agent contacts the remote DF agent to request and compare this information. Following this procedure the migration is usually led to success.

No other alternatives to publish the service are considered, since this is the only method available within the IEEE-FIPA specifications. Technical details for the service registration are provided in Appendix A.1.

Main Migration Protocol

An agent migration is initiated when an agent sends a specific ACL message to the AMM local to the migrating agent. An agent can request the migration of itself or another agent. A specific ontology for this message is defined in Appendix A.2. After this message is received the AMM initiates the Main Migration Protocol (MMP).

MMP, which is implemented by the AMM agent, manages the whole migration process (Figure 3.4). MMP is composed of an agent interaction protocol, which drives the part of the process common to all the agent migrations, and the PS component. This component is a sequence of three steps implemented by switchable migration protocols (Section 3.4). These steps provide specific functionality and personalise each agent migration according to the protocols selected by the agent that has requested the migration. These protocols can provide migration authentication and several ways of transferring agents among others.

MMP starts the process with the exchange of a first message between the local and remote AMMs involved in the migration. The information contained in this first message includes (Table 3.1): the Migration Identifier (MID), the Agent Identifier (AID), the CGID, the agent data and state encoding algorithms, the agent profiles (one or more depending on the number of associated agent codes and their corresponding CIDs), the agent version, and the migration protocols to use in the PS component. An agent migration is only possible if the two involved agent platforms have, at least, one transfer protocol in common, they agree on the protocols to use, and there is, at least, an agent profile supported. Otherwise, the migration process is refused. If the protocols selected did not agree, another protocol selection can be chosen by the agent, and the migration can be retried from scratch. The migration process concludes when all the steps are carried out and the last message of the interaction protocol has been sent.

MMP is implemented using a custom agent interaction protocol called Synchronized Request (refer to Appendix A.3 for details), which is defined using Agent UML

Elements			Description
migration-id			Migration Unique Identifier (MID).
name			FIPA Agent Identifier (AID).
cgid			Agent code group identifier (CGID).
agent-profile	cid		Agent code identification associated to the profile.
	system	name	The name of the mobile agent system environment.
		major-version	Major version.
		minor-version	Minor version.
		dependencies	Dependencies required.
	language	name	The name of the mobile agent language.
		major-version	Major version.
		minor-version	Minor version.
		format	Code base format of the mobile agent.
		filter	Filter to execute over the code base before execute.
		dependencies	Dependencies required.
	os	name	The name of the operating system.
		major-version	Major version.
		minor-version	Minor version.
		hardware	Hardware below the operating system.
		dependencies	Dependencies required.
data-encoding			Data encoding method.
state-enconding			State encoding method.
agent-version			Agent version.
pre-transfer			Pre-transfer protocols chosen.
transfer			Transfer protocol chosen.
post-transfer			Post-transfer protocols chosen.

Table 3.1: First message content.

[BMO01, OPB01]. It is based on the issue of two consecutive ACL message requests in a single protocol, where the second request is only sent in case of the first's success (left side of Figure 3.4). These procedure is consistent with the model proposed in Section 3.3.1, where the agent is transfered and, later, its execution resumed. The use of a single interaction protocol contributes to have a compact MMP with all the mandatory migration operations grouped altogether.

Individual agent migrations are uniquely identified, in the context of the two involved APs, by a string called MID. The MID is included in the *reply-with* and *in-reply-to* fields of all the ACL messages exchanged during the process (the first field is used in the messages sent from the origin and the second in the messages sent from the destination). In some cases the MID is also included as part of the message content.

A specific ontology defined in Appendix A.4 is used for all the ACL messages exchanged in MMP. The ontology defines the actions associated to the two *request* messages of MMP (move/clone and resume respectively), and several data structures.

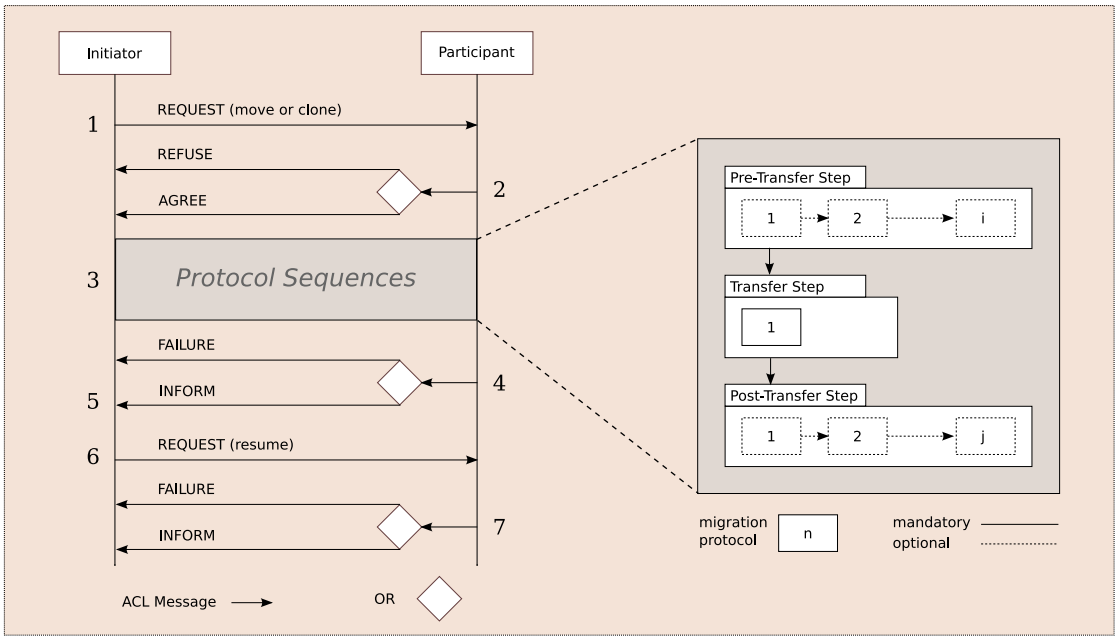

Figure 3.4: Main Migration Protocol.

The functionality of MMP is achieved by following the process listed above, which constitutes the immutable part of the architecture. The specific steps of the protocol can be seen on the left side of Figure 3.4 and are detailed in the following lines:

1. A first message (Table 3.1) with the agent profile, requirements, and a list of protocols feasible to be used within PS is sent.

2. The request may be refused or agreed by the remote AM, regarding whether the agent requirements are satisfied and the requested protocols supported.

3. All the agreed protocols in the first message are executed in the corresponding steps (right side of Figure 3.4) in the PS component.

4. The agent is registered in the remote platform (i.e., in the AMS agent of IEEE-FIPA compliant AM). Then, a message is sent to the origin to notify the success or failure of the agent registration.

5. If the agent is successfully registered, then it is removed from the local platform.

6. A message is sent to resume the agent execution in the remote platform.

7. A message is sent to the origin to notify the success or failure of this last operation.

In case of error, the process between the two involved AMMs is cancelled. This is done by using the corresponding interaction protocols methods (refuse or failure messages) or, in case of not being possible, by using the FIPA-Cancel-Meta-Protocol defined within the FIPA Request Interaction Protocol [FIP02k]. The error type is indicated by the corresponding exception predicate (defined in Appendix A.4), included as content of the message, according to Section 3.3.3.

Protocol Sequences

At the right side of Figure 3.4, there are represented the three steps that characterise each migration: Pre-Transfer, Transfer and Post-Transfer. These steps customise the migration process by using specific migration strategies (in the agent transfer), authorisation protocols, resource negotiation, or whatever kind of process agreed between the involved platforms. They are represented as the three rectangles of the figure, and implemented by the set of protocols chosen in the first message sent in MMP. They are drawn inside the mentioned rectangles as dotted line boxes. Nonetheless, there is an exception with the Transfer step. Unlike the other ones, a protocol, and only one, must be selected. In this case the protocol is represented as a single continuous line box to denote its mandatory usage. Steps and selected protocols are executed in the order indicated by the arrows. The steps are:

- **Pre-Transfer**: Contains protocols that are run before the agent transfer, e.g., extra parameters negotiation, authorisation, resource agreement, among others. They are usually negotiation protocols.

- **Transfer**: Contains a protocol to transfer the agent code, data, and state. Different protocols support different migration strategies (push, on demand, etc.). One and only one protocol can be assigned to this step.

- **Post-Transfer**: Contains protocols that are run after the agent transfer, e.g., integrity check, validation of code signatures, agent results transfer, among others.

Protocols used in these steps are defined and standardised apart from the main architecture. Each one of these protocols is characterised by having a well-known name; by having a well defined functionality suitable for the associated step; by being independent from the rest; by being composed of one or more stages implemented by interaction protocols; and by optionally using one or more message content ontologies. In Section 3.4 several transfer protocols are presented.

3.3.3 Error Management

The migration process defined by IPMA, which is described in Section 3.3.2, might be subject to network transmission errors, specially if unreliable network protocols are used, e.g., protocols based on UDP. The following lines present an evaluation of the error treatment in terms of protocol analysis at the communication and middleware levels.

The communication level deals with ACL Messages and agent interaction protocols. In the following lines there is a review of the most common communication errors, and the measures adopted for their treatment in the present architecture:

- **Message unexpectancy**: Messages with a performative and/or conversation identification that are not expected. These messages are simply ignored, since they do not fit with the expected ones and they are not processed by any interaction protocol.

- **Message corruption**: Message corruption is detected at a higher level. Depending on the corrupted message part the effects could be different. If the message basic information is damaged, such as the envelope or headers, the message might be discarded. Thus, the error would be treated as a message loss. In some other cases, if the message is not discarded, a `not-understood` message is issued. On the other hand, if the message content is damaged, the migration process between the two AMMs could be cancelled.

- **Message duplication**: Duplicated messages are considered by the interaction protocols as unexpected messages. In case of duplication of the initiator message (the first sent by an interaction protocol), the duplicate is also accepted by the receiver. But in this case, it is detected at the migration architecture level by checking the MID included in the message with the registers of current agent migrations.

- **Message loss or delay**: Lost messages, can be detected by setting an expiration time as indicated in the FIPA Request Interaction Protocol specification [FIP02k]. The action to do in this case regards to the specific implementation. The use of timeouts is strongly recommended, although not mandatory. Message timeout has to be carefully chosen to avoid confusing message delay with message loss, and to supply enough time to process large messages.

- **Out of order delivery**: Because of the nature of the interaction protocols used in MMP, messages cannot be received out of order, since a message is not sent without the acknowledge of the previous one. Even though, in case some specific protocol needs to send several messages without waiting for confirmation, the possibility of out of order messages must be taken into account, e.g., FrTP of Section 3.4.

The middleware level deals with exceptions thrown by some of the actions requested to the remote location during the agent migration process, e.g., agent code registration, agent platform registration, agent execution resumption, and so forth. When an exception is thrown a message with the `failure` performative, which includes an error predicate as a message content, is sent. The predicate indicates the type of error and contains a description of it. For more information of each predicate, check the ontology of each protocol in Appendixes A.2, A.4, A.5 and A.6. In some cases errors can also be thrown in the local platform. Therefore the remote platform is notified that the process is cancelled from the initiator part using the FIPA Cancel Meta Protocol described in [FIP02k].

The complete error management is carried out by the AMM agent. Errors lead to the interruption and cancellation of the dialog between the two involved AMMs. The default policy in case of unrecoverable error is restarting the process (MMP) carried out

by the two involved AMMs. Thus, it is responsibility of the AMM to retry the agent migration without bothering the agent which has requested it.

In the Figure 3.5 there is an example of an agent migration that fails in its first attempt. The AMM agent can be programmed to resume the local agent execution, if the migration cannot be successfully done, after a limited number of retries. Besides, a failure message with an error description is sent to the agent which has requested the migration. According to the error description the agent might choose to change the migration parameters and retry it on its own. A number of migration retries can be suggested in the request message sent by the migrating agent to the AMM (Appendixes A.2).

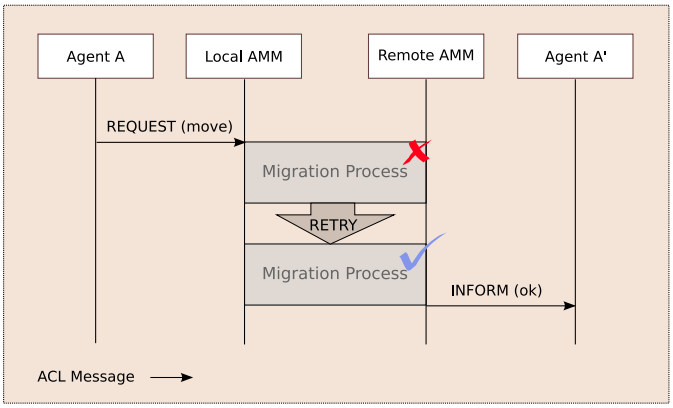

Figure 3.5: AMM Error Management.

Notice that the AMM agent must keep a persistent copy of the migrating agent during the whole migration process. Otherwise, in case of a failure after the agent is killed the local agent execution could not be resumed. This approach is similar to other ones presented in the literature [SBS00, LCW04] providing fault tolerance in the agent migration.

3.4 Mobility Protocols

The protocols presented in this section are managed from the PS component of IPMA. Depending on the step where a protocol belongs to (Pre-Transfer, Transfer or Post-Transfer), different aspects of the migration process are characterised.

The transfer protocols define the migration strategy followed in the migration. The migration strategies of MASs characterise the way agent codes are managed in the migration process. Depending on the migration strategy the agent may have different properties. In [BR05] two main groups of migration strategies are distinguished: push and pull.

- **Push**: all the agent code is sent together with the agent data, and state. This strategy gives the agent the property of autonomy, since it does not depend on any other resource present in previous locations (Figure 3.6).

- **Pull**: the agent code is requested from the destination location once the agent data has reached it (Figure 3.7). This strategy has two variants, the pull-at-once, and the pull-on-demand. The first one, the pull-at-once, requests all the agent resources from the destination locations once the agent migration request reaches it. The second one, the pull-on-demand, requests the agent code as needed during the whole life of the agent. This last variant keeps the agent associated to the location that hosts its code. Therefore, it reduces the agent autonomy, although it can provide better performance in some cases.

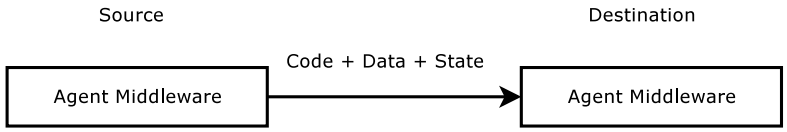

Figure 3.6: Push migration strategy.

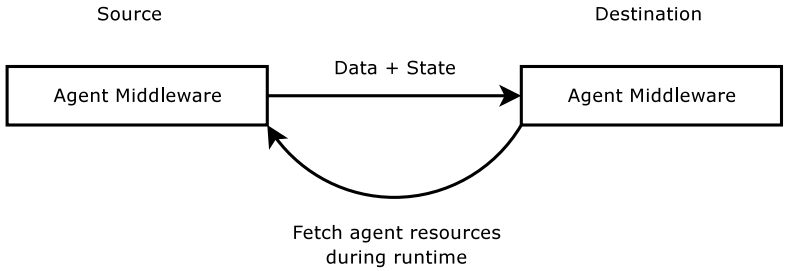

Figure 3.7: Pull migration strategy.

The migration strategies described above are the basics to other ones. Taking them into account several transfer protocols (PCTP, ODTP, FrTP, and RESTTP) for the PS Transfer step of IPMA are described in the following paragraphs.

3.4.1 Push Cache Transfer Protocol

The Push Cache Transfer Protocol (PCTP), as the name states, is a protocol to transfer the agent code, data, and state based on the push migration strategy. It supports the use of a code caching mechanism that prevents from transferring agent code that is already present in the destination platform. Its internal name, used to announce and select it, is `pctp-v1`.

This protocol is addressed to agents that need to be independent of their home agent platform, since all their resources (code, data, and state) are always carried with them. Because of the ACL transport cost (Section 3.4.5), it is not indicated for agents with a large quantity of resources.

Protocol's operation

The process to send the agent code, data, and state is divided in two parts (Figure 3.8). In the first part, the agent data and state, which are always needed, are sent to the destination platform together with the MID, the CID, the SR, and the HCID. A response is sent according to the existence or not of the agent code. If the code is present there is no need to carry out the protocol's second part. Then, it finalises its execution saving up two ACL messages. Otherwise, the second part is effectively run to send the code.

The protocol is composed of two FIPA Request interaction protocols, one for each part, and a specific ontology (Appendix A.5) that defines the associated actions to each request and the mentioned information included in each message. Errors are managed in the same way as in MMP (Section 3.3.3), although with specific exception predicates regarding this protocol. An example of a typical migration in terms of ACL messages is shown in Figure 3.9.

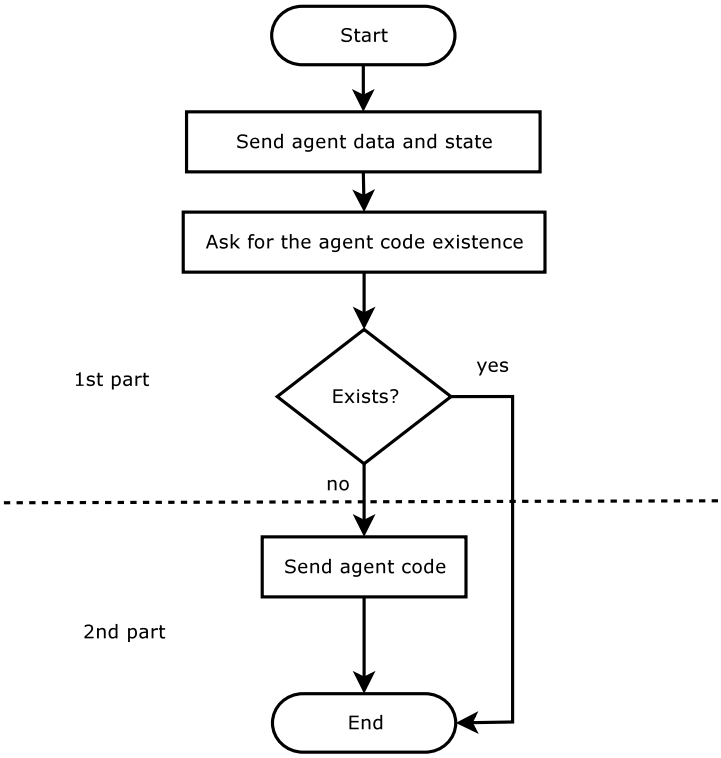

Figure 3.8: Push Cache Transfer Protocol diagram.

3.4.2 On-Demand Transfer Protocol

The On-Demand Transfer Protocol (ODTP) is a protocol to transfer the agent code, data, and state according to an on-demand pull strategy. First, only the data and state are transfered, and when a code snippet or some other agent resource is needed, it is individually requested to a resource server. A list of resource servers is specified by the agent which requests the migration. Any AM may adopt the role of an agent resource server. The internal name of the protocol is `odtp-v1`.

This protocol is addressed to agents that do a partial use of their code in each visited location. Only the agent data and state components, in addition to a list of the other necessary resources, are sent in each migration. Then, only the needed code is requested from each location. As a consequence, agents cannot be unbound from their home agent platform or from the specific servers which maintain their code. Furthermore, since each code request introduces a delay, the protocol is only recommended for short-distance

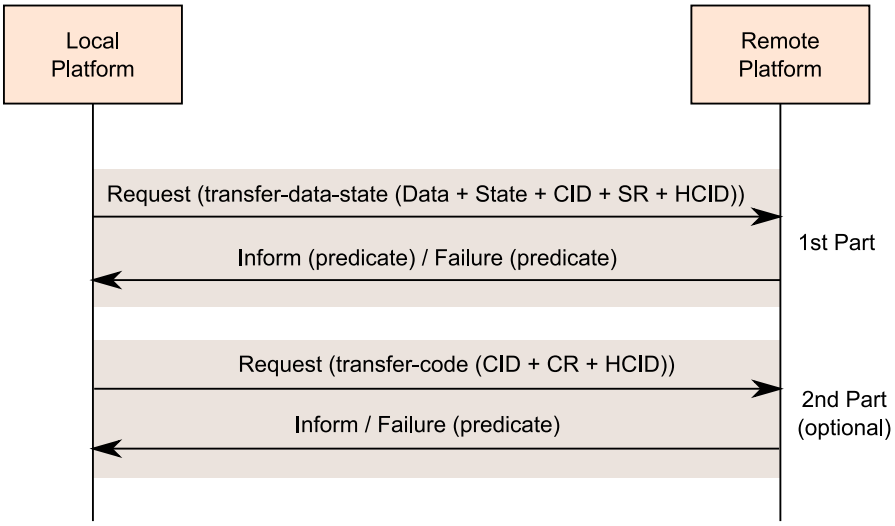

Figure 3.9: Push Cache Transfer Protocol flow diagram.

migrations.

Protocol's operation

An agent is composed of several resources (classes, libraries, images, and so on). These resources must be identified in some way, otherwise they could not be requested to the resource server because they would be unknown. To overcome this issue, an agent resource list is included in the first message exchanged within the transfer protocol (see an example in Figure 3.10). The resource list contains a tuple for each agent resource. Since this protocol is not related to a specific PL and/or system, the constraints about resource identification regarding each PL and/or system are unknown, e.g., multiple resources could have the same name although being different. For this reason each tuple includes the agent resource name together with a cryptographic hash of it, e.g., an entry can be the name of a Java class with its corresponding MD5 hash. Therefore, the exchange of migration strategies during the agent life is straightforward, since all the agent resources are known and can be easily gathered, prepared, and packaged for a different strategy, e.g., an agent that arrives to a location using the ODTP protocol, and that leaves it using the PCTP protocol.

Two components are involved in the use of this protocol. On the one hand, there is

Hash Algorithm: *md5*	
Resource Name	**Resource Hash**
jipms.test.MobileAgent	*2a6cdf4c817d1567f10acef26c16f009*
jipms.test.MobileAgent$Behaviour	*f58322f88787d1ece7df4f309d45744a*
jipms.test.AgentGUI	*46909c0978614e8d5df9a6384e1d83db*
jipms.test.AgentItinerary	*355554767010531a33af172242a086b2*
jipms.test.AgentLocation	*4ac00a67400b5dc3191e0f7889032b01*

Figure 3.10: ODTP Resource List example.

one component part of the PS transfer step that sends the basic agent information. And on the other hand, there is another component running stand-alone that remains active waiting to serve resource requests as long as they are needed by the agents. When an agent is sent using this protocol, the first component sends the agent state, data, resource list, list of resource servers, MID, CID, SR, and HCID (left side of Figure 3.11). And, when the agent needs some resource, e.g., a code snippet, it is individually requested and served by the second component (right side of Figure 3.11). Resources are usually requested to the home agent platform, although they may be in other locations. A list of possible locations to request agent resources is included in the first message exchanged.

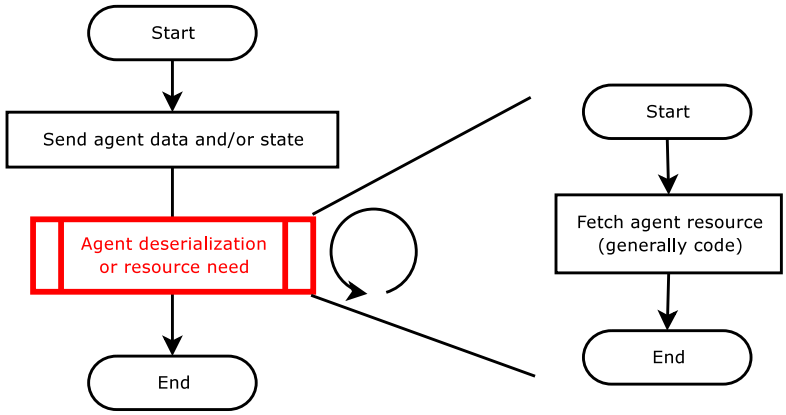

Figure 3.11: On-Demand Transfer Protocol diagram.

The protocol is also based on the IEEE-FIPA Request interaction protocol. Two specific ontologies (Appendix A.6) define the actions for the two possible requests (transfer the agent and fetch resources), and define the information contained within the ACL

messages. Errors are managed in the same way as in MMP (Section 3.3.3), although with specific exception predicates regarding this protocol. An example of a typical migration in terms of ACL messages is depicted in Figure 3.12. Notice that the second block of exchanged messages is repeated as many times as it is needed regarding the number of requested resources. In the example these resources are requested to the local agent platform, although in a real system they are requested to the resource server specified by the migrating agent.

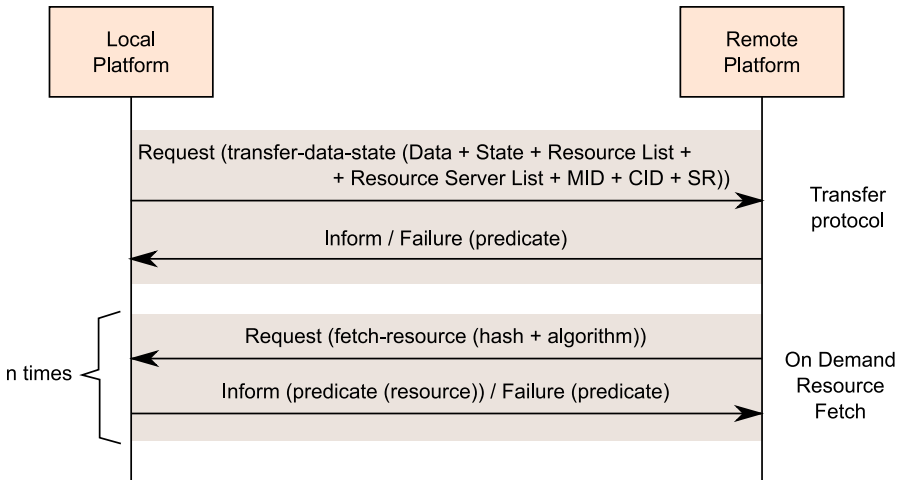

Figure 3.12: On Demand Transfer Protocol flow diagram.

3.4.3 Fragmented Transfer Protocol

The Fragmented Transfer Protocol (FrTP) is a protocol to transfer the agent code, data, and state over several ACL messages. It has been devised to avoid sending too much data into a single ACL message as PCTP does. FrTP is also based on the push migration strategy, and it also supports code caching mechanisms. Its internal name is `ftp-v1`.

This protocol, such as PCTP, is addressed to agents that need to be independent of their home agent platform, since all their resources (code, data, and state) are always carried with them. Nevertheless, PCTP encapsulates all the agent data, and state, and all the agent code into two ACL messages. Since these messages are not conceived to transfer large quantity of data, several implementations may obtain a bad performance as the message size increases (there are several studies of the agent communication

performance using ACL messages, see [Cuc04] and [JJK06]). This is the reason why in this protocol the data and code are fragmented in several snippets, with a fixed size, which are encapsulated into different ACL messages.

Protocol's operation

The protocol can be divided into three parts (Figure 3.14). The first of them negotiates the parameters to transfer the agent components, sends the agent state if it is applicable, and embraces the other two parts. In the first message exchanged there are several parameters and the agent state (which is not fragmented). The parameters are the desired size of the fragments used to transfer the agent code and data, the code size, the data size, the MID, the CID, the SR, and the HCID. A response is sent refusing or accepting the agent transference. If the remote location does not agree in one or more parameters (agent size, fragment size, and so on) it is refused. Otherwise, it is accepted and it is specified if the agent code is needed or not (take into account that a code caching mechanism may exist in the destination location). If the code is present, only the agent data fragments are sent.

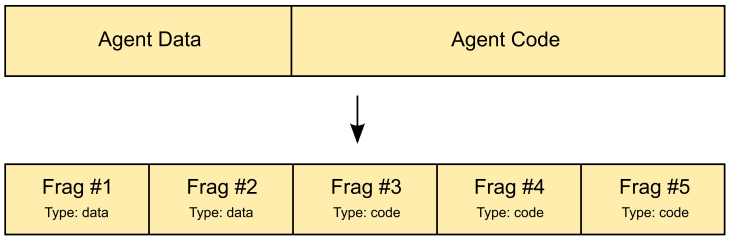

Figure 3.13: Agent components fragmentation.

The second part of the protocol is the fragment's transference. This is a simple part where the agent data and code are sent in several fragments (Figure 3.13). Each fragment, in addition to a piece of data, includes a unique identification, a type of fragment (a string indicating if the fragment contains code or data), and the MID. The corresponding messages are sent as they are produced, no acknowledgements are expected between each message sent.

The third part is a fault tolerance mechanism to recover lost fragments. If one or more fragments have not been received after a fixed period of time they are specifically

requested (it is like a negative acknowledge). This part only has sense if no reliable transport protocols are used to transfer the ACL messages. It has the advantage of not increasing the protocol time in most of the agent transmissions, since it is only used when one or more messages have not been received. The information sent to request the lost fragment is the same associated to the original one (the fragment identification, the fragment type, and the MID). The fragment requested on demand is encapsulated in the same type of message than the fragments previously sent. Finally, the whole protocol finalises by sending a message informing about the success or failure of the operation (this message belongs to the first part of the protocol).

The protocol is composed of two FIPA Request interaction protocols, one for the first part and another for the third part, and a specific ontology (Appendix A.7) that defines the actions associated to each request and the mentioned information included in each message. Errors are managed in the same way as in MMP (Section 3.3.3), although with specific exception predicates regarding this protocol. An example of a typical migration in terms of ACL messages is illustrated in Figure 3.15.

3.4.4 REST Transfer Protocol

The REST Transfer Protocol (RESTTP), as the name states, is a protocol that transfers the agent code, data, and state combining the Representational State Transfer (REST) technology [FT02], which is a coordinated set of architectural constraints based on the standard HTTP protocol, with ACL messages. It is proposed as an alternative protocol to PCTP since REST is more suitable for the transfer of high amounts of data than the ACL messages. The RESTTP is based on a pull-at-once migration strategy, since the needed resources are requested by the remote location using HTTP requests. Therefore, the support for code caching mechanisms is implicit, if some resource is not needed it is not requested. Its internal name is `resttp-v1`.

This protocol is addressed to agents that need to be independent of their home agent platform, since all their resources (code, data, and state) are always carried with them. Thanks to the HTTP data transport efficiency it is specially indicated for agents with a large quantity of mandatory resources, i.e., the ones required in any location visited.

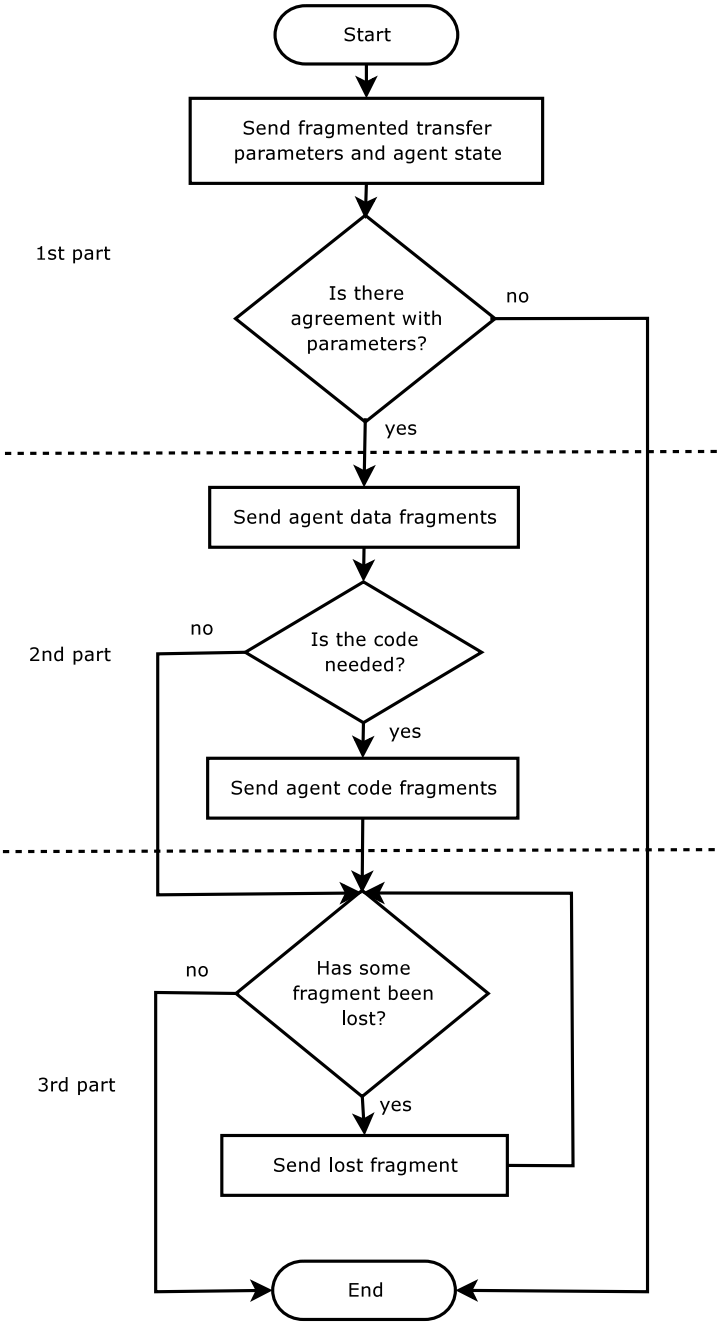

Figure 3.14: Fragmented Transfer Protocol diagram.

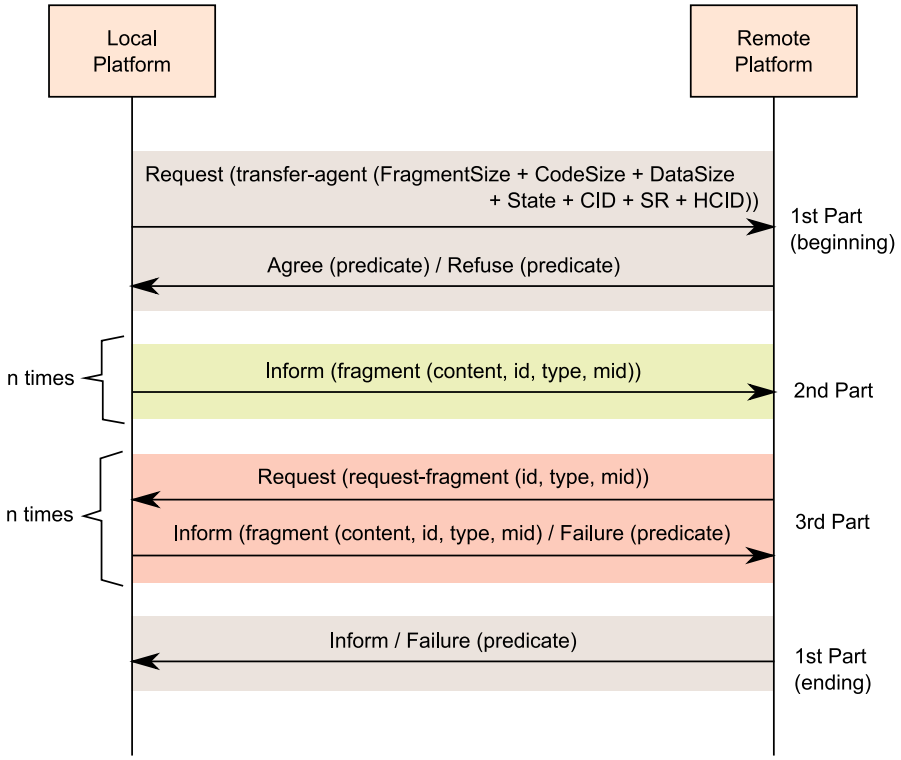

Figure 3.15: Fragmented Transfer Protocol flow diagram.

Base URL:	*http://server/resttp-v1/{mid}/{nonce}*				
Element	**Method**	**Request Param.**	**Headers**	**Body**	**Functionality**
{nonce}	GET				Get the agent resource requested.

Table 3.2: RESTTP Interface.

Protocol's operation

The process to send the agent code, data, and state is composed of two parts, one driven by ACL messages where several parameters are exchanged, and the other driven by HTTP requests where the agent components are transfered. The agent code, data, and state, are served by an HTTP server under a random unique name (*nonce*) only valid during the transaction. A random name is used to prevent external parties from unfairly downloading the agent resources. Furthermore, it is recommended to remove each resource once it has been downloaded as means as an only once usage policy.

The protocol starts by putting the agent code, data, and state into the HTTP server, and sending an ACL message with the random names assigned to them, in addition to the host and port of the HTTP server, and the MID, CID, SR, and HCID identifiers (Figure 3.16).

Once this message has been received, the destination location requests the agent code, data, and state, as convenient, using several HTTP requests (one for each agent resource, see Table 3.2).

Then, after getting the agent resources, an inform or failure ACL message is sent from the remote location to finalise the protocol. When this message is received, the source location removes the resources from the HTTP server (some of them may already be removed if the only once usage policy has been applied).

The protocol is composed of one FIPA Request interaction protocol, one simple REST interface (Table 3.2), and a specific ontology (Appendix A.8) that defines the associated actions to the request message and the information included in each message. Errors are managed in the same way as in MMP (Section 3.3.3), although with specific exception predicates regarding this protocol. An example of a typical migration in terms of ACL messages and HTTP connections is shown in Figure 3.17.

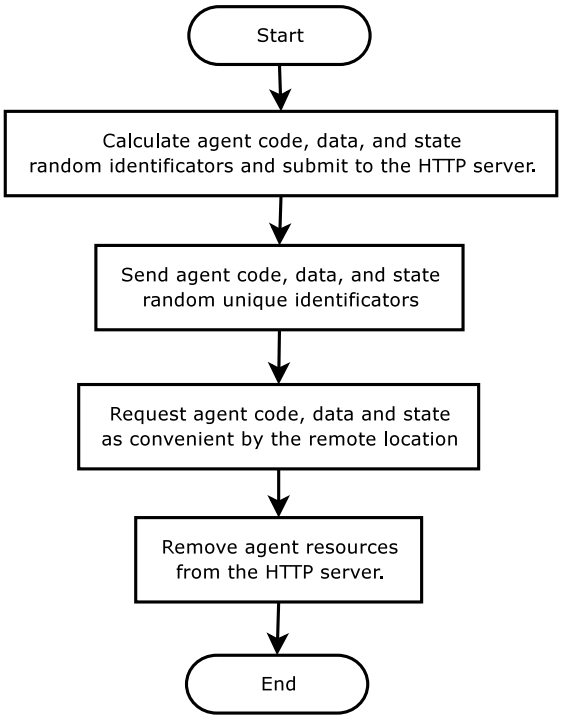

Figure 3.16: REST Transfer Protocol diagram.

3.4.5 Protocols usage

In this section four protocols have been presented. Three of them (PCTP, FrTP, and RESTTP) are based on transferring all the agent components before resuming the agent execution. The other one (ODTP) transfers the agent data and state in advance, but requests the agent code as needed.

Since not all the protocols are appropriate for all situations, in several cases the best strategy is to combine them, i.e., in one agent movement use one protocol and in another one use the other. An example of this is an agent with several code components that moves through many communities with many nodes with a high network delay between them. In this case, a good strategy is to use the push migration (PCTP or FrTP) to move between communities (moving all the agent code through the wide area network) and, then, use the on-demand migration (ODTP) for agent movements within the community (requesting only the needed code in the local area network). In this example, the number of messages exchanged between long distant locations is minimised. The

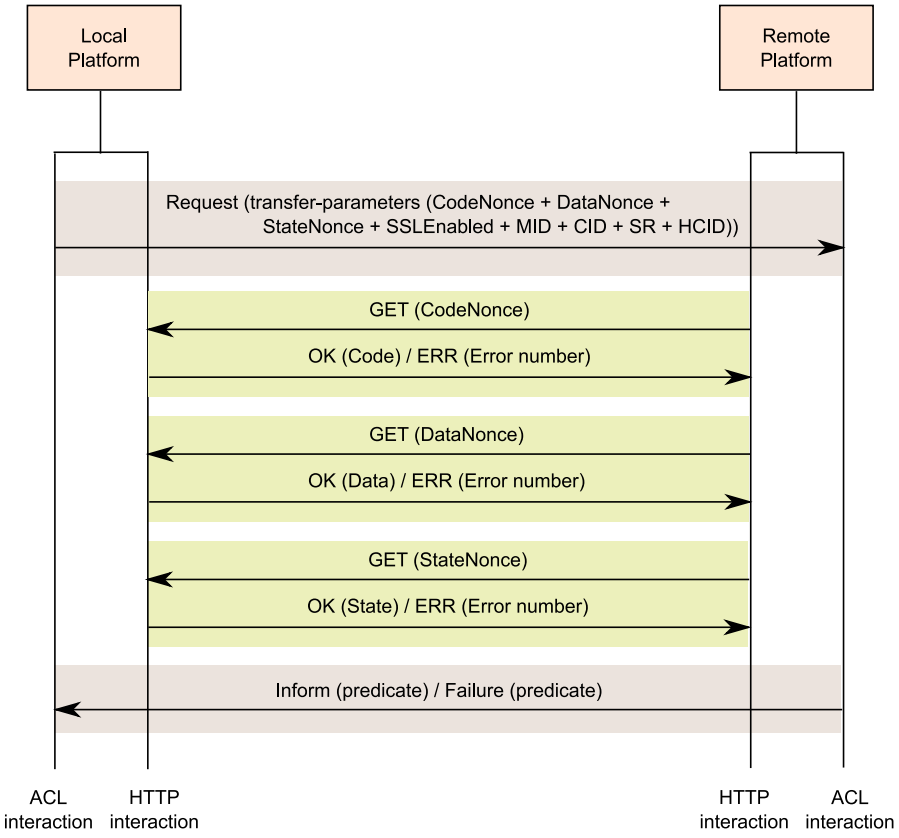

Figure 3.17: REST Transfer Protocol flow diagram.

mentioned protocols combination is technically feasible. The implementation presented in Section 3.6 considers this possibility.

The election between PCTP, FrTP, and RESTTP depends on the agent components size, and the requirements to only use IEEE-FIPA standards in the agent transference. The first two are not so efficient as the last one, although they use only these standards. And regarding these two, FrTP is the more efficient for agents with large components. A detailed analysis of their performance is presented in Chapter 4.

3.5 Mobility services comparison

Several agent mobility models are implemented in the existing mobile AMs. In this section a comparison is done between these mobility models, in most cases strongly

tightened to their middlewares, and the one presented in this chapter (IPMA).

IPMA stands out by being independent of specific AMs and/or PLs, and by relying on the well-know IEEE-FIPA agent standards. Furthermore, it offers a flexible migration process that can be individually customised for each agent migration, providing several agent migration strategies. It is also important to notice that chosen migration options are agreed or refused using the first exchanged messages. Thus, it is not possible to send an agent without agreement of the involved parts. This is specially significant in low bandwidth links, which might be charged according to the amount of data transfered, like the General Packet Radio Service (GPRS) in cellular networks.

The rest of the section compares the mobility services of the most representative MASs. A summary of the comparison is presented in Table 3.3 which evaluates the following features:

- **Mobility service name**: The name of the mobility service.
- **Mobile Agent System**: The name of the reference MAS where the service is implemented.
- **Agent in the first message**: It states if the agent is directly sent in the first message exchanged or there is some kind of pre-negotiation.
- **Exportable model**: It indicates if the migration model is designed only for the AM analysed or if it can be used in another ones.
- **Migration strategies**: The migration strategies supported by the specific implementation of the model.
- **Migration initiation**: It indicates who initiates the migration process.
- **Mobility type**: The types of mobility supported, referring to weak or strong mobility.
- **Language supported**: The PL supported by the model.
- **Transport**: The underlying transport mechanism to transfer the messages or data used in the agent migration.

- **Security**: The security features implemented.
- **Agent standards**: The agent standards used by the mobility service.

The *Aglets server middleware* [LM98] is an environment to execute Aglets, which are Java Mobile Agents (MAs), developed by the IBM Tokyo Research Labs. It supports autonomous execution of Aglets, dynamic routing itineraries, and it is compliant with the MASIF specifications. Although this AM is currently not active, it is described for historical reasons and because of its complete migration system. The mobility service is based on a protocol called Aglet Transfer Protocol (ATP), which is only used in this AM. The protocol is similar, in structure, to the Hyper-Text Transfer Protocol (HTTP), although the request messages are adapted to migration operations. In Aglets, agent migrations can be initiated from the same place where the agent resides or from another one. In this last case the process of requesting the agent return is called Aglet Retraction. The migration strategy used is a combination of the push and pull strategies. When the agent migrates the in-use agent classes and the specified JAR files are transfered in a single step (push). But if extra code is needed, it can be requested (pull) on demand to the agent code base (this is a list of servers included in the agent to locate its code). Despite the fact that the ATP protocol offers a flexible migration strategy, since the push and pull strategies can be combined, IPMA is more open and can support new strategies proposed in the future. Regarding the agent migration initiation, IPMA accepts migration requests for a specific agent from any other agent, even if they are sent from different locations. In fact, it is similar to the Aglet Retraction method.

Agent Operating System (AOS) [vNOT+07] is the mobility specification used within the AgentScape [OB04] and Mansion [vNBT04] AMs. AgentScape is a multi-language mobile AM designed to support scalable, secure, distributed multi-agent applications. Agents migrate between virtual domains called *locations* and can negotiate resource requirements before migrating [MOB06]. Mansion is a mobile AM strongly focused on security. AOS, which is the lower layer in these two AMs, provides low-level secure communication between the high level middleware processes, between agents, and secure agent mobility using a set of SunRPC methods. A pull migration strategy is used, so all the agent data and code are requested from the destination location all at once. PL dependent run-time environments for agents are provided through different agent

servers. This solution is PL independent and, even, it could support strong mobility under certain circumstances. Comparing the AOS mobility specification with IPMA, our solution is not limited to one migration strategy and takes advantage of the IEEE-FIPA well-known agent standards. Although IPMA does not provide agent resources negotiation nor security, they can later be incorporated as specific protocols within the PS steps.

Security-centric Mobile Agent server (SeMoA) [RJS01] is a runtime environment for Java-based MAs focused on security and easy extendibility. Its migration service provides secure agent mobility based on extensible filters that allow to sign, encrypt, or carry out any cryptographic operation related to the agent transmission. All the agent resources (code, data, life cycle state...) are grouped in a structure called "AgentContext", which is sent all at once over a network protocol (http, raw, raws...) that can be chosen before migrating. If part of the code is not present in this structure, it can be requested to the agent code base. The migration strategy, which is the same strategy of the Aglets AM, is a combination of a push and an on-demand pull strategies. Although it is not possible to change this behaviour, the migration strategy of SeMoA is more flexible than the one presented in the previous AM, but not as much as IPMA. Moreover, the only supported PL is Java and, therefore, weak migration is the only possible one. Our architecture, on the other hand, is not bound to a specific PL and migration strategy. Finally, although the security features present in SeMoA are not provided in IPMA, they could be incorporated as specific protocols in the PS steps.

The *Kalong* [BR05] architecture, implemented as a software module, is focused on providing an efficient mobility service to the Tracy [BR05] AM. Nevertheless, it is generic enough to be used in other AMs, e.g., in the JADE AM [PBK05]. Tracy [BR05] is a modular, component-oriented, extensible Java AM designed as a micro-kernel. The aim behind this platform, according to its authors, was to provide a toolkit usable *for the development of industrial-strength real-world applications*. Its mobility service, Kalong, uses the Simple Agent Transmission Protocol (SATP), which defines a set of binary messages to support all the common migration operations (it is similar to MASIF [OMG97], which provides a set of CORBA IDL methods to perform these operations). The advantage of this model is that the migrating agent combines these

operations making up any migration strategy. Although Kalong provides a well-defined flexible migration system, which allows the use of different agent migration strategies, it has the disadvantage of being confined to the Java PL. Compared with IPMA, Kalong has a higher performance and it is a bit more flexible. Nevertheless, IPMA is based on consolidated agent standards (IEEE-FIPA), it is not limited to only one PL, it might support weak and strong migration, and the migration request can be done by any agent.

Mobility service	**ATP**	**AOS**	**SeMoA**	**Kalong**	**IPMA**
Mobile Agent System	Aglets	AgentScape, Mansion	SeMoA	Tracy, JADE	JADE, AgentScape
Agent in the first message	Yes	No	Yes	No	No
Exportable model	No	Yes	No	Yes	Yes
Migration strategies	Push + Pull On-Demand combination	Pull	Push + Pull On-Demand combination	Any combining SATP messages	Any implementable
Migration initiation	Any agent	Oneself	Oneself	Any agent	Any agent
Mobility supported	Weak	Any	Weak	Weak	Any
Language supported	Java	Any	Java	Java	Any
Transport	Proprietary.	SunRPC	Any	Binary	FIPA-MTP
Security	Authorisation	Encryption	Authorisation, Encryption, others.	Any	Any
Agent standards	MASIF	No	No	No	IEEE-FIPA

Table 3.3: Migration services comparison.

3.6 JADE Inter-Platform Mobility Service

The migration architecture described in Section 3.3 and the protocols of Section 3.4 have been implemented in the JADE AM. JADE has been the first AM chosen to test the proposed migration model because it is the most widespread nowadays and it is IEEE-FIPA compliant.

The service implemented provides weak inter-platform mobility, allowing agents to migrate between different platforms. The referred implementation is published as a development release of the JIPMS add-on [JIPa].

3.6.1 JADE Introduction

JADE [BCG06] is an IEEE-FIPA compliant AM. It provides a fully distributed system where agents can reside and move between different *containers*. The JADE containers are an abstraction to spread the platform over multiple hosts (Figure 3.18). A central main container takes care of the other ones' management. The agents defined by IEEE-FIPA reside in the main container.

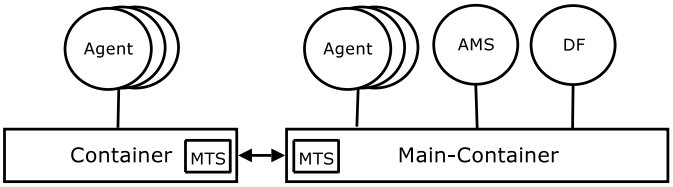

Figure 3.18: The JADE agent middleware architecture.

Because of the distributed nature of the platform, a complex services architecture is used [Cai04]. This architecture, thanks to its flexible structure based on vertical and horizontal commands, makes possible the addition of new services to the platform without modifying it. Moreover, it has the benefit of allowing interaction and collaboration between other services. This architecture is very flexible, but it is also quite complex, because it has to manage all the platform services and coordinate all their instances spread in the JADE containers. The underlying communication between containers is done by using Java RMI calls.

JADE has a default mobility service, based on the JADE services architecture, that only allows migration between containers, what is known as intra-platform mobility. Conversely, the service implemented according to IPMA, JIPMS, allows the migration of agents between different agent platforms, i.e., it offers inter-platform mobility. Use of both services can be combined at any time of the agent life.

Agent tasks in JADE are implemented as behaviours. A JADE behaviour is the abstraction used in this AM to represent agent tasks. Behaviours, once are created, are scheduled for execution in an agent. Each agent is assigned to a Java thread which is shared by its behaviours. The scheduling of the agent behaviours is cooperative, i.e., the next behaviour scheduled is not executed until the previous one has finalised its task. To prevent some behaviours from blocking the other ones, agent developers can divide the

tasks in several subtasks, each one individually scheduled (these tasks can be perceived as if they were running concurrently). Nevertheless, in cases where true concurrency is needed, the ThreadedBehavior is used. It is a special type of behaviour which has assigned a specific thread.

3.6.2 JIPMS Basics

JIPMS [JIPa] is a complex piece of software, specially taking into account that it coexists with the default JADE intra-platform mobility service built into the platform. In the last revision of JIPMS all the agent transfer protocols proposed in Section 3.4, PCTP, ODTP, FrTP, and RESTTP, have been included.

JIPMS has been implemented at the application level, as it is specified by IPMA, but taking advantage of several components included at the middleware level. The main reason is that JIPMS has been developed to guarantee a perfect integration with the JADE intra-platform mobility service and the JADE containers. Therefore, privileged access to the AM is required. Otherwise this integration would be impossible. Nevertheless, all the components implemented at middleware level do not require AM modifications, thanks to the advanced JADE services architecture [Cai04].

3.6.3 JIPMS Structure

JIPMS, as previously mentioned, is implemented between the middleware level, in the context of the JADE services architecture, and the application level, in the context of the AMM agent. JIPMS is composed of several components that are explained in the following paragraphs.

At middleware level, as can be appreciated in Figure 3.19, JIPMS is composed of three main components: the Mobility Service, the Code Manager, and the Class Analysis Library. These components are replicated in each JADE container since an inter-platform migration can start from any of them. Between the middleware and the application levels there is the Agent Platform Accessor component. And at the application level there is the AMM component, which is part of IPMA. The AMM is only present in the main container since IPMA is defined in terms of the IEEE-FIPA standards, which

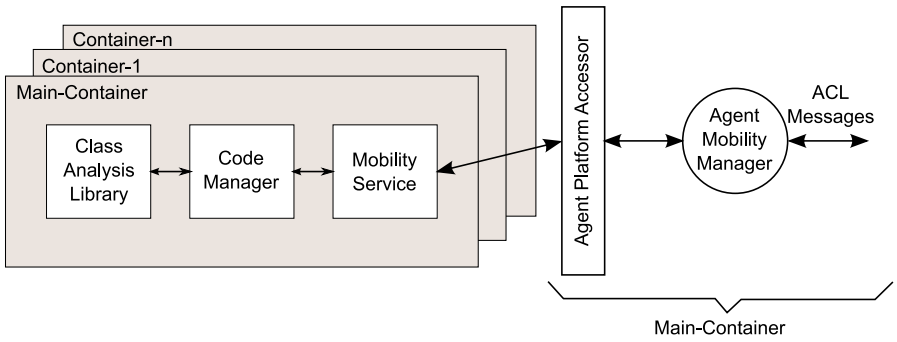

Figure 3.19: Inter-Platform mobility service parts.

do not consider the JADE container abstraction. Nevertheless, the direct migration from the other containers to another AP is possible, although in this case the ACL message exchange is always done from the main container. The operation which is not allowed is directly migrating an agent to a non main container of another AP.

Mobility Service

The mobility service is the JADE service component that deals with the initialisation of JIPMS and the coordination of actions that involve several JADE containers. Most of these actions are requested from the application level by the AMM.

Code Manager

The code manager is a generic repository where the code used by MAs is registered and maintained. It is composed of a list that binds specific agents with their codes, and specific managers to maintain different types of code, e.g., there is the JarManager to maintain codes packaged into JAR files, and the ClassManager, to maintain Java classes individually packaged. There is another component, called CodeLocator, in which we have actively collaborated that belongs to the JADE platform and binds specific MAs with their specific Java class loaders. The CodeLocator and the Code Manager make possible the future addition of new mobility algorithms based on different ways of fetching and loading the agent code without modifying the agent platform, and guaranteeing

the compatibility with the intra-platform mobility service.

Class Analysis Library

In the JADE AM the agent's code is not packed into any specific format, and it is not separately managed from the JADE code. On the contrary, it is simply included in the Java class path. Agent migrations require gathering the agent code to send it to remote locations. JIPMS provides a manual and an automatic methods to select and package the code of a MA.

The manual method consists of creating a JAR file with all the agent code with the file name following the pattern `PackageName_AgentMainClass`. This file is put into a specific folder wherefrom JIPMS retrieves it. The automatic method is based on the Class Analysis Library (CAL), which provides several tools to analyse the agent main class and recursively gather the agent dependent classes collection. Dependencies are found by analysing the class constant pool tables [LY99], which include low level class information.

CAL first searches for direct dependencies of the main agent class, then it searches for dependencies of these direct dependencies, and so on. Several dependencies are omitted because they are also found in remote locations, e.g., the String class. Otherwise, unnecessary code would be sent.

Agent Platform Accessor

The Agent Platform Accessor (APA) is a component used by the AMM as a gateway to reach the agent platform facilities, e.g., get agent code, data, and state, register an agent, start an agent, and so forth. It preserves the AMM agent from dealing with too many implementation details. Therefore, it is a bridge between the middleware level components and the application level components.

Agent Mobility Manager

The Agent Mobility Manager (AMM) is the only mandatory component according to IPMA. AMM implements all the protocols used in the migration process and can be

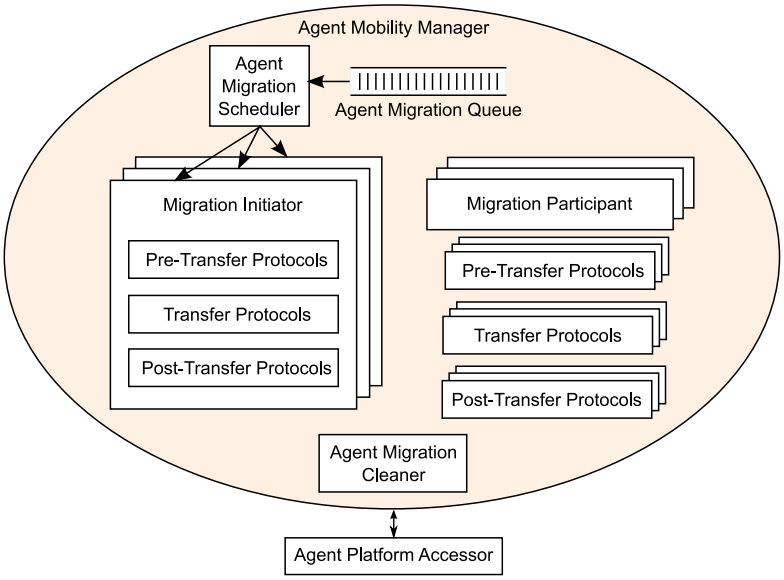

Figure 3.20: Agent Mobility Manager.

considered as the front-end of the mobility service. All the service code independent of the specific AM should be implemented in the AMM agent.

In this implementation, as it is depicted in Figure 3.20, the AMM agent is composed of an Agent Migration Queue, an Agent Migration Scheduler, an Agent Migration Cleaner, a Migration Initiator and a Migration Participant components, which are implemented as JADE behaviours. Since AMM deals with several agent migrations, some components must be replicated and require concurrency. Then, several replicated behaviours exist. In case of components with blocking processes, which could stop the execution of the other behaviours, they are implemented as ThreadedBehaviours.

When an agent migration is requested an entry is enqueued to the Agent Migration Queue. The Agent Migration Scheduler is in charge of dealing with these enqueued migration requests and assign them to a non busy Migration Initiator. Several migrations can be concurrently attended because multiple instances of most of the AMM components are instantiated. The Migration Initiator and the Migration Participant components implement the MMP, one processing outgoing migrations and the other processing incoming migrations. The protocols associated to the IPMA PS (pre-transfer, transfer,

and post-transfer protocols) are executed within these components. Finally, there is the Agent Migration Cleaner, which deals with cleaning tasks of interrupted or lost migrations.

3.6.4 JIPMS Protocols

According to IPMA, each migration process is driven by MMP. There are also a set of other protocols executed within the PS component which characterise the agent migration.

These protocols are subject to change in the future and new ones can be included. Thus, they are developed following a specific interface which facilitates their design and integration within the service. A JIPMS protocol, according to the mentioned interface, is composed of a name, a type, a set of behaviour initiators, a set of behaviour responders, and a set of ontologies. The name is used in the selection of the protocol and the type indicates if it is a pre-transfer, transfer, or post-transfer protocol. Behaviour initiators and responders are matched in stages, e.g., a protocol with only one stage or part will have one initiator behaviour and one responder behaviour. Usually these behaviours implement an interaction protocol. And one ore more ontologies are provided in order to represent the content of exchanged messages.

The protocols implemented are detailed in the next paragraphs. In all the cases only weak migration is implemented (the agent state is not sent), the agent data is encoded according to the Java serialisation mechanisms, and all the protocols can be combined during the agent lifetime, allowing agents to choose the most convenient migration strategy at each time.

Push Cache Transfer Protocol

The PCTP implementation is composed of two pairs of Initiator/Responder behaviours, one ontology, the JarManager component, and the JarClassLoader. The agent code exchanged is packed into a JAR file.

The behaviours and the ontology simply implement the process described in Section 3.4.1. Nevertheless, this is not sufficient to deal with the agent code transfered. It

is necessary to manage received codes and to have a class loader to load them.

The JarManager is the component which manages the code. This component indexes the code according to its unique identification, which in this implementation is calculated from the MD5 hash of all the agent classes. A cache mechanism based on this identification and on a code reference counter has also been included to avoid maintaining replicated JAR files. Other tools are implemented in the JarManager, such as the automation of agent code gathering using the Class Analysis Library, and the generation of JAR files from other code managers, e.g., to make a JAR file from an agent received by ODTP. A JAR class loader has also been implemented, which is currently integrated with the standard distribution of JADE.

On Demand Transfer Protocol

The ODTP implementation is composed of two pairs of Initiator/Responder behaviours, two ontologies, the ClassManager component, and the OnDemandClassLoader. The agent code, in this case, is exchanged as single Java classes.

The behaviours and ontologies simply implement the process described in Section 3.4.2. As in the previous protocol, this is not sufficient to deal with the agent code transfered. In this case it is necessary to have a class loader which is in charge of requesting the code required at each moment. Furthermore, the management of this code must also be taken into account.

The OnDemandClassLoader implements the part of the protocol that fetches the agent code. This code is managed by the ClassManager component. Two main functionalities are associated with this component. On the one hand, it is used to serve the agent code. Agents using ODTP can freely select any platform of their itinerary to act as a server of all their code. Therefore, they can migrate to other locations and request their code to one of the platforms which the agent has established as code servers. On the other hand, the component implements a cache mechanism which maintains the classes which have been previously fetched during a specific period of time for later reuse.

Fragmented Transfer Protocol

The FrTP implementation is composed of several behaviours and an ontology. Furthermore, it takes advantage of the JarManager and JarClassLoader components developed for PCTP. The agent code exchanged is also packed into a JAR file.

The implementation has been focused on getting the best possible performance, rather than getting an easily understandable and comprehensible development. This is the reason why there are four behaviours with several functions assigned to each one. The first two deal with the first part of the protocol (Figure 3.15) and, furthermore, the responder behaviour also deals with the treatment of lost fragments corresponding to the third part of the protocol. The reason is that the two parts involve the treatment of a request message, although with different actions, and it is more efficient to deal with them from the same behaviour. The other two behaviours are in charge of sending and receiving the code and data fragments. The responder part also takes care of controlling the reception of all the required fragments. In case of fragment loss, this behaviour individually requests the fragments lost (requests are attended by the first pair of behaviours described).

Finally, minor changes have been done to the JIPMS add-on in order to optimise the protocol performance, e.g., the agent code is directly registered using a JAR file path instead of reloading a Java byte array with all the code.

REST Transfer Protocol

The RESTTP implementation is composed of one pair of Initiator/Responder behaviours, one ontology, one HTTP server, one HTTP client, one REST interface, and the JarManager and JarClassLoader components used in PCTP. The agent code exchanged is also packed into a JAR file.

The behaviours and ontology simply implement the transfer negotiation process described in Section 3.4.4. The HTTP server and client are in charge of transferring the agent resources using the REST interface. Regarding the REST part of the protocol, it has been implemented using the Grizzly HTTP Web server and the Java API for RESTful Web Services (JAX-RS) [HS07] specification, which is implemented by the Jersey Java project. All the data management is based on the use of Java streams, therefore it

is handled as efficiently as possible.

3.6.5 JIPMS Usage

The mobility service currently presented has been integrated with the default JADE mobility service. Agents migrate between containers and between platforms by using the *doMove(location)* method of the agent base class. The agent migration request through ACL messages is not implemented yet.

The destination of an agent migration is indicated by using an object implementing the Location interface. The type of object used determines the mobility service chosen. The interface is implemented by the ContainerID class, if it is used for the intra-platform migration service, and by the PlatformID class, if it is used for the inter-platform migration service.

In case of inter-platform mobility, moreover, the agent can select the protocols to migrate and some particular parameters regarding them. This is achieved by using several methods provided in the Helper of the mobility service, a specific service management class available to all the agents. Refer to [Cai04] for more information about the JADE service architecture.

3.7 AgentScape Inter-Platform Mobility Service

The migration architecture described in [COO+07], which is a preliminary version of IPMA described in this thesis, and a push transfer protocol have been implemented in the AgentScape AM (Section 3.7.1) as ASIPMS. AgentScape has been selected to do a validation test [COO+07] of the IPMA model in non IEEE-FIPA compliant AMs (Section 3.7.2). This validation test has permitted to evaluate the general difficulties present in this kind of AMs to implement the mentioned model (Section 3.7.4).

This implementation has been kept simple (Section 3.7.3). Only a push migration strategy has been included. In this case the agent code and data are packed as in the previous implementation, by means of a Java JAR file and by means of the Java serialisation mechanisms respectively. Since, also in this case, only weak migration is supported, the agent state is not included. The referred resulting implementation is called ASIPMS.

3.7.1 AgentScape introduction

AgentScape [OB04] is a secure multi-language mobile AM. Agents migrate between virtual domains called *locations* (Figure 3.21). An AgentScape location consists of one or more hosts running the AgentScape AM, typically within a single administrative domain.

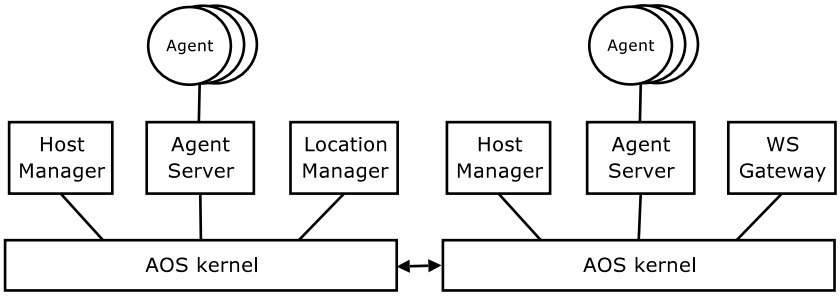

Figure 3.21: The AgentScape middleware architecture.

The default migration service of AgentScape is based on a pull-at-once migration strategy. The interaction between the involved platforms uses a set of SunRPC methods defined within the context of the AOS Kernel (see Section 3.5). This kernel is a secure subsystem below the AM that manages communications and agent containers. It is worth taking into account that in AgentScape an agent container is a special package with the agent code, data, and other resources.

3.7.2 FIPA Message Transport Service

IPMA relies on the use of the IEEE-FIPA specifications for the implementation of the different protocols that compose the migration process. The AM for which the implementation is targeted should support at least a minimum set of the IEEE-FIPA standards:

- The agent naming scheme [FIP04], thus migrating agents must have a FIPA compliant identification.
- ACL messages must be supported [FIP02e], since the whole migration architecture is based on the ACL message exchange.
- A Message Transport Service (MTS) to manage the ACL messages is required.

- A shared content language must be defined (e.g., SL [FIP02l]) to be able to interpret the data exchanged inside the ACL messages.

- Interaction protocols (Request and Propose), since they are used in several protocols of the architecture.

AgentScape does not support these specifications. Therefore part of this functionality has been ported from the JADE AM. Nonetheless, the Message Transport Service (MTS) has been implemented from scratch (reusing the MTP-HTTP of JADE, implemented by Exposito *et al.* [EARL03]).

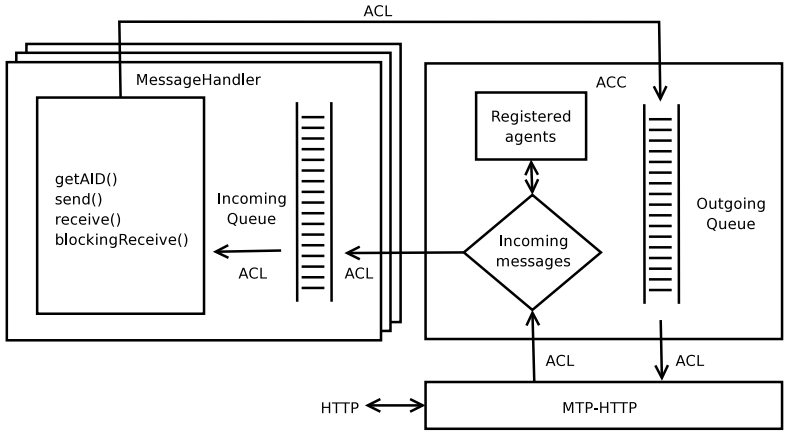

Figure 3.22: AgentScape FIPA MTS.

The implemented MTS, as shown in Figure 3.22, is composed of three components: the MTP-HTTP, the ACC (Agent Communication Channel), and the MessageHandler.

- **MTP-HTTP**: Module that encapsulates ACL messages over the HTTP protocol, as defined in the IEEE-FIPA specification [FIP02f].

- **ACC**: Main core of the messaging service. It is mainly composed of an ACL message outgoing queue and a registry of agents.

- **MessageHandler**: Component used by agents to send and receive messages. Each agent has its own instance because it contains the agent's ACL message incoming queue.

The only requirement for agents using this messaging system is to be registered to the ACC with an IEEE-FIPA compliant name (AID). Then, they obtain a MessageHandler to interact with the service.

Furthermore, the FIPA Request and FIPA Propose interaction protocols have also been implemented. The Request and Propose interaction protocols are developed as abstract classes, providing several methods which are triggered in reaction to each event of the protocol. Since the concept of agent behaviour does not exist in AgentScape, threads are used to implement some of the agent tasks.

3.7.3 ASIPMS Structure

ASIPMS is completely implemented at the application level within the AMM agent. The AgentScape AM has not been modified at all. The AMM agent, in this case, is composed of: an Agent Platform Accessor, an Agent Migration Queue, an Agent Migration Scheduler, a Migration Initiator, a Migration Participant, and the migration protocols (pre-transfer, transfer and post-transfer steps). There is no Class Analysis Library, since AgentScape agents are already packed in a special structure called Agent Container which includes all the agent code required by the agent. Therefore, the Class Analysis Library functionality is not necessary.

The AMM is implemented and registered as a standard AgentScape agent. Moreover, it is registered to the IEEE-FIPA MTS to allow communication with other remote AMM agents. Once started, the AMM remains waiting for agent requests to migrate, and for ACL messages of other AMMs to start the reception of other agents. Migration protocols are implemented with the IEEE-FIPA interaction protocols provided by the IEEE-FIPA MTS previously explained.

3.7.4 Open issues

There are some open issues when IPMA, described in Section 3.3, is used in no IEEE-FIPA compliant AMs. In the next list there are the most important issues and the adopted solutions in this implementation:

- **IEEE-FIPA compliance workaround**: The set of IEEE-FIPA standards required

for the migration architecture has to be added to the AM. For simplicity reasons, they have been integrated at the middleware application level. This option is the least complex because it can be implemented using static libraries and agents without modifying AM internals.

- **IEEE-FIPA agent naming**: Agent local naming scheme should be mapped to the IEEE-FIPA agent naming scheme and vice versa. In case direct mapping between the two schemes were not possible, a method based on tables linking local and IEEE-FIPA agent names had to be developed to do name translations. Currently, there is no automatic name scheme translation implemented in AgentScape.

- **Middleware local agent naming**: Local agent name (name following the AM local scheme specification) could be maintained in the destination platform, e.g., an AgentScape agent which migrates to another AgentScape AM do not only could keep its IEEE-FIPA name, but it could keep its AgentScape name. Nevertheless, IPMA does not guarantee these local names. In fact, agents using a mobility service based on this architecture always must refer to each other agents using their IEEE-FIPA compliant names, since the other location could be a different type of AM. Thus, in the ASIPMS implementation the local agent name is not kept.

- **Agent life-cycle**: The agent life-cycle shows the operational state of an agent at a specific moment. By using it, for example, the messaging service can decide whether deliver a message to an agent or wait to do it later. When an agent migrates, its life cycle should be in a specific state (the Transit state in IEEE-FIPA). Therefore, the agent life-cycle provided by the AM, using the migration proposed, should be in an equivalent state regarding to the IEEE-FIPA life cycle. This might be specially important when in a nearby future agents are migrating between different kinds of AM (Chapter 6).

This implementation is developed in the context of a validation test of IPMA in non IEEE-FIPA compliant environments. Hence, only a subset of the solutions proposed have been implemented since not all the issues are critical. Nevertheless, they have to be taken into account for future implementations.

3.8 Conclusions

In this section an agent migration model based on the IEEE FIPA agent standards, IPMA, has been presented. It is a complete, robust, flexible, sufficient, and minimal model.

- The model is complete, since it supports different migration strategies and mobility types, it is independent of any specific AM and PL, and it is application level oriented.

- The model is robust, since the task of migrating an agent is delegated to a specialised agent which follows a set of well-defined steps. There is no possibility of misunderstanding, like in other migration models, such as Kalong, where the migrating agent has the responsibility to drive the migration. Notice that the order of the basic operations is immutable to minimise the model complexity. Another consideration to the model robustness is that the execution of a migrating agent is never resumed before it has been unregistered from the source location. Therefore, the model is collision free, since an agent cannot go back to the source location before being unregistered from there.

- The model is flexible, since the subprocesses executed in PS (Figure 3.4) vary according to the migration protocols selected by the migrating agent. This mechanism is the key to obtain customised agent migrations, from the most simple ones, with one transfer protocol, to the most complex ones with many other protocols in use, e.g., different migration strategies, agent authentication, and so on. Nevertheless, this flexibility does not compromise the model robustness in anyway, since the migration is always managed by the AMM.

- The model is sufficient, since it covers all the needed operations to do a successful migration. MMP and all the associated steps of PS are entirely executed in each agent migration.

- The model is minimal, since it exchanges the minimum number of messages to follow the migration model proposed in Section 3.3.1.

The validity of the mobility model presented has been contrasted by a theoretical comparison with other mobility services, and by the empirical implementation in real AMs. As a result, the JIPMS for JADE and the ASIPMS for AgentScape have been developed. In the next chapter, the results of a set of performance tests are presented. Several theoretical assessments regarding the migration strategies are demonstrated with the practical results.

Chapter 4

Performance Analysis

This chapter presents a set of tests to evaluate the most relevant performance differences between the migration transfer protocols proposed for the Inter-Platform Mobility Architecture (IPMA): the Push Cache Transfer Protocol (PCTP), the On-Demand Transfer Protocol (ODTP), the Fragmented Transfer Protocol (FrTP), and the REST Transfer Protocol (RESTTP).

4.1 Introduction

IPMA, described in the previous chapter, is devised to support several migration strategies and features. In addition to the architecture, four migration transfer protocols have been proposed (see Section 3.4) to provide migration strategies suitable for different environments.

In this chapter a set of tests has been carried out to compare the protocols' performance implemented in the JADE Inter-Platform Mobility Service (JIPMS). The chapter has been divided in six parts. Firstly, Section 4.2 describes the evaluation setup used in the tests. Then, Section 4.3 compares PCTP and ODTP. Section 4.4 compares PCTP and FrTP. And, Section 4.5 compares PCTP and RESTTP. Different scenarios are described for each of these comparisons. Finally, in Section 4.6 a discussion and usage recommendations for the migration protocols analysed are presented.

4.2 Evaluation setup

The evaluation setup used to run the tests are two Pentium IV at 2 GHz, with 778 MB of RAM, and a GNU/Linux based operating system (Fedora Core 5 distribution) with kernel version 2.6.17, both with a dedicated 100 Mb/s switched Ethernet network. The Agent Middleware (AM) used is the JADE 3.5 with the development version of the JIPMS add-on [JIPa] and the FIPA Message Transport Protocol (MTP) HTTP. For RESTTP the external Jersey 0.7 libraries are used. Several network environments are simulated using the NetEm [Hem05] Linux utility over the mentioned network. For resource consumption reasons, each Agent Mobility Manager (AMM) agent is limited to process 20 incoming and 20 outgoing concurrent agent migrations. In case of more migration requests, they are queued until one or more processes have finalised.

The performance tests have been carried out by using a, specifically created, agent mobility test suite [JIPb]. It is composed of a programmable agent that creates a set of agents which follow a specific itinerary (list of locations) a fixed number of times (called iterations from now on). When the agents finalise their itinerary the selected number of times, they send a message to the programmable agent notifying it. Finally, the average time consumed by each agent migration round-trip to the AMs is calculated:

$$Average\ time = \frac{Total\ time}{Iterations * Number\ of\ agents}$$

The concurrent migrations are processed following the philosophy of a pipeline. In pipelines, the individual time to produce a product is longer than the average time spent per product. Therefore, the results shown in the result tables, for the case with several agents, must be seen as a measure of throughput for the AMs.

The itinerary performed in the next tests by the agents, which consists of two locations, is repeated ten times or one hundred times depending on the type of agent tested. The process is repeated many times to minimise the AM and agent start-up time influence over the results. Furthermore, each test is performed five times and the final results are product of the average outcome [MFB+07]. Usually the tests are carried out with 1, 10, or 100 agents concurrently migrating. This allows to analyse the average time that each of these agent migrations consumes to the AM.

4.3 Performance evaluation 1: PCTP vs ODTP

This first part of the performance evaluation comprises the PCTP and the ODTP migration protocols. Two kinds of tests have been done, one with lightweight agents (small code with only two Java classes), and another one with heavyweight agents (large code with twelve or thirty-two Java classes). Each test has been performed, with 1, 10, or 100 agents running simultaneously. The LAN environment described in the evaluation setup (Section 4.2) has been chosen to perform the tests, since it is the most appropriate for ODTP. PCTP is latterly tested in other environments.

Therefore, the parameters modified in the tests are: the number of agent classes each agent has, the number of iterations or round-trips each agent does, and the number of agent instances concurrently running. It must be taken into account that in PCTP the agents' code is encapsulated in a compressed JAR file. While in ODTP the code is not compressed and it is always served from the first location of the itinerary (therefore it is local to one of the platforms, although it is always locally requested when the cache mechanisms are disabled).

4.3.1 Lightweight agents

The first set of tests uses a lightweight agent to follow the itinerary between the two locations. A lightweight agent is composed of two Java classes which weigh 4KB in total. Since protocols implemented provide code caching mechanisms, two versions of the tests have been performed, one with this feature enabled and the other one with it disabled.

The results can be appreciated in Table 4.1. Several facts can be stated with the help of Table 4.2 and Figure 4.1. First of all, the on-demand strategy with cache mechanisms disabled consumes more time than the push strategy. In this case, with the push strategy 9 messages are used in front of the 11 messages used in the on-demand strategy (take into account that the code is composed of two classes, and two messages are required to request each class). If cache mechanisms are enabled the messages used are the same, 7 messages in this case, and, therefore, the time spent is similar for both.

The use of code caching mechanisms improves the migration performance, since

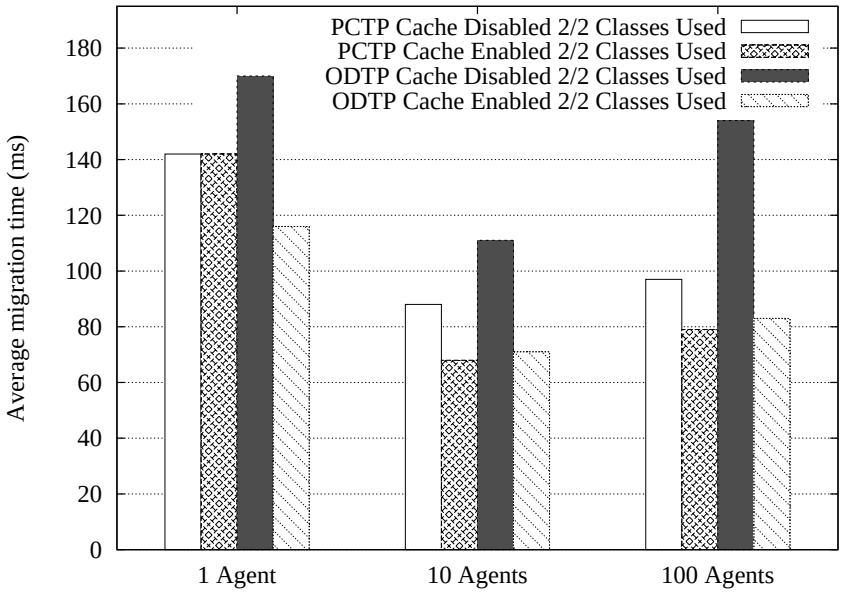

Figure 4.1: PCTP vs ODTP 2 Classes.

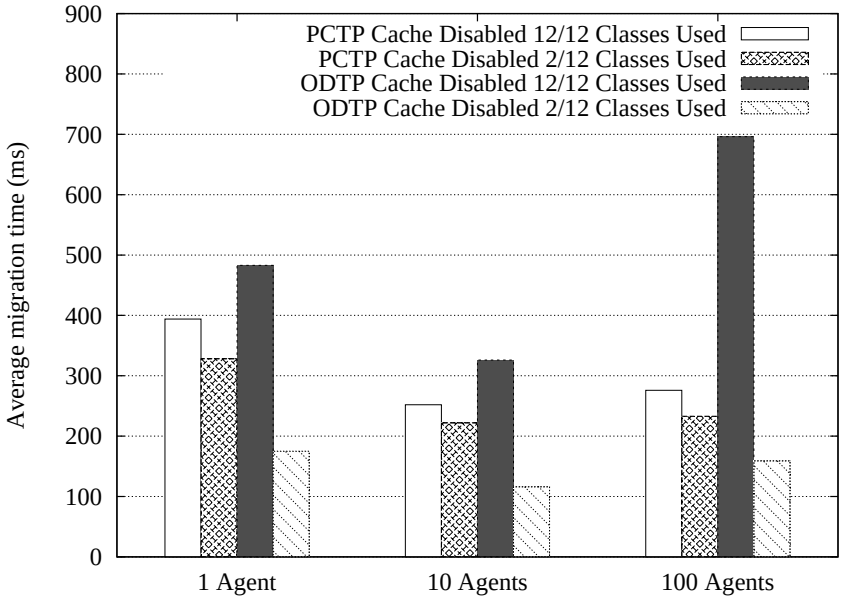

Figure 4.2: PCTP vs ODTP 12 Classes.

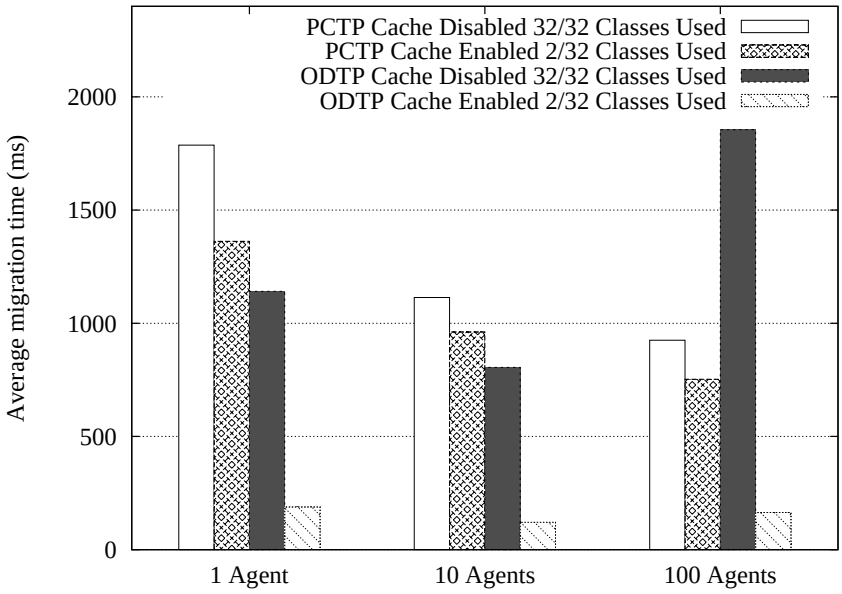

Figure 4.3: PCTP vs ODTP 32 Classes.

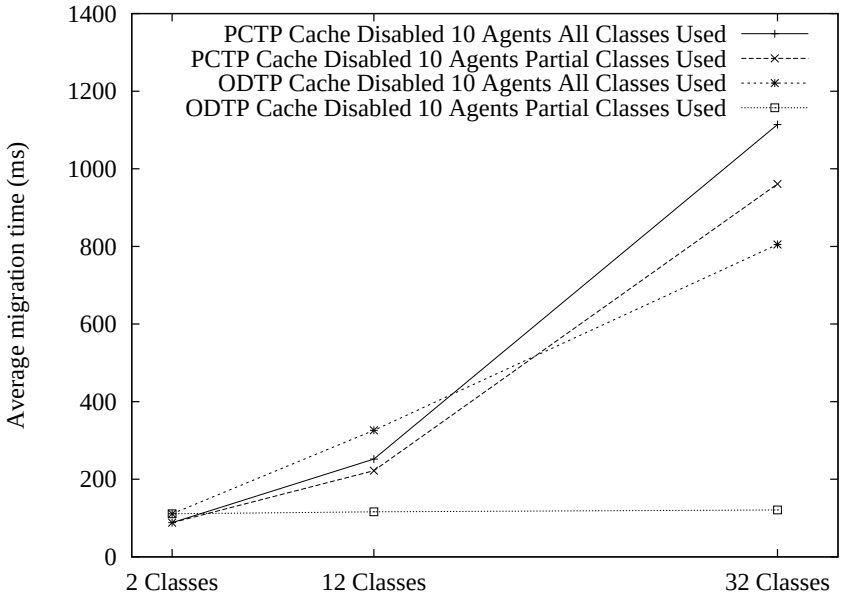

Figure 4.4: PCTP vs ODTP 10 Agents.

	Code cache?	1 Agent	10 Agents	100 Agents
Push migration	No	142	88	97
	Yes	142	68	79
On Demand migration	No	170	111	154
	Yes	116	71	83

Table 4.1: Lightweight agents migration performance (in ms).

	1 Agent	10 Agents	100 Agents
PCTP Cache usage (t.decrease)	0 %	22.73%	18.56%
ODTP Cache usage (t.decrease)	31.76%	36.04%	46.10%
PCTP vs ODTP No-Cache (t.decrease)	16.47%	20.72%	37.01%
PCTP vs ODTP Cache (t.decrease)	-22.41%	4.23%	4.82%

Table 4.2: Lightweight agents migration comparison.

the agent code is only transfered once (there are only two locations, therefore one code transfer is enough). The benefits of these mechanisms are more noticeable with ODTP. The reason is that in PCTP two messages are saved up regarding ODTP, where two messages are sent for each requested agent class. There is an exception for the case of only one agent using PCTP. The PCTP cache mechanisms only maintain the agent code while it is being used by at least one agent. Therefore, when there is only one agent migrating, the code is not maintained in the cache after the agent leaves the platform, and it cannot be reused the next time the agent arrives. This is the reason why the results in this case are exactly the same with and without the cache mechanisms activated. Furthermore, when the results are compared with ODTP, this last takes advantage of the mentioned implementation drawback and gets a better average round trip time.

It can also be appreciated that one migrating agent consumes, in average, more time than several ones (do not confuse this with the real time spent in each agent migration from the agent perspective). The reason is that ten agents migrating simultaneously are pipelined without having to wait for the others' migration processes finalisation. Migration processes are parallelised. Nevertheless, when one hundred agents are launched, the average migration time increases regarding the ten agents test. There are three reasons:

- more agents means more load in the AM and in the migration service.
- more agents means more migrations and, therefore, more messages exchanged

and more load in the message transport service.

- JIPMS has been set up to 20 maximum concurrently migrations, then the remaining 80 migrations have to wait in a queue. Increasing this maximum means more performance but more consumed resources too.

4.3.2 Multi-class Heavyweight Agents

The second set of tests use heavyweight agents composed of many classes to follow the itinerary between the two locations. There are two types of agents, one with a code that weighs 39KB and it is composed of 12 Java classes (results are shown on Table 4.3), and another one with a code that weighs 108KB and it is composed of 32 Java classes (results are shown on Table 4.5). Two of these classes belong to the agent core, the rest are extra classes to increase the agent weight, each one of 3KB. Each protocol is tested with the two mentioned types of agents (12 and 32 classes), each one with two versions, one using all the code (12/12 and 32/32 classes) and another one using only part of them (2/12 and 2/32 classes).

The tests are only performed with the code caching mechanisms disabled. The reason is that the main difference between the protocols analysed is the way the code is transfered. Therefore, code caching mechanisms reduce the code transfer to only one (the first migration). In this case the results do not present significant differences between the two protocols (compare, for example, the last two rows of Table 4.2).

From Figures 4.2 and 4.3, and Tables 4.3 and 4.5, several facts can be stated. First of all, the time spent in an agent round trip increases as the agent code size grows. Using ODTP this is more noticeable, since more classes means not only more data to transfer, but more messages to exchange. Furthermore, it must be taken into account that PCTP sends the agent code within a JAR file which is compressed, transferring only 26KB and 72KB respectively for the two agent types, but spending more CPU time.

The next paragraphs discuss the statements got from comparing the two migration protocols (Table 4.4 and Table 4.6). Regarding PCTP, the outcomes of tests performed by agents using all the code, 12/12 and 32/32 classes, are only slightly different from the ones got by agents using only part of the code, 2/12 and 2/32 classes (see first and

	Classes used	1 Agent	10 Agents	100 Agents
Push migration	12-12	394	252	276
	2-12	328	222	233
On Demand migration	12-12	483	326	696
	2-12	175	116	159

Table 4.3: Heavyweight agents (12 classes) migration performance (in ms).

	1 Agent	10 Agents	100 Agents
PCTP part vs all code (t.decrease)	16.75%	11.90%	15.58%
ODTP part vs all code (t.decrease)	63.77%	64.42%	77.16%
PCTP vs ODTP all code (t.decrease)	18.43%	22.70%	60.34%
ODTP vs PCTP part code (t.decrease)	46.65%	47.75%	31.76%

Table 4.4: Heavyweight agents (12 classes) migration comparison.

	Classes used	1 Agent	10 Agents	100 Agents
Push migration	32-32	1,787	1,114	925
	2-32	1,362	961	753
On Demand migration	32-32	1,141	805	1,855
	2-32	188	121	164

Table 4.5: Heavyweight agents (32 classes) migration performance (in ms).

	1 Agent	10 Agents	100 Agents
PCTP part vs all code (t.decrease)	23.78%	13.73%	18.59%
ODTP part vs all code (t.decrease)	83.52%	84.97%	91.16%
PCTP vs ODTP all code (t.decrease)	-56.62%	-38.39%	50.13%
ODTP vs PCTP part code (t.decrease)	86.20%	87.41%	78.22%

Table 4.6: Heavyweight agents (32 classes) migration comparison.

second rows of Table 4.4 and Table 4.6). The push migration strategy always sends all the agent code, it does not matter whether this code is used or not in the remote location. Despite this, agents that only use two classes perform slightly faster migrations, since they do not load the rest of them in the main memory.

Regarding ODTP, in case of using all the agent code (12/12 or 32/32 classes), which is the worst case for this strategy, the results are rather higher than using PCTP. When one hundred agents are migrating, the time spent is very high. Twelve or thirty-two classes are requested for each agent, which means exchanging 24 and 64 messages respectively just to transfer the agent code, in front of the 2 messages used in the push migration strategy (although larger than the others). These results confirm the statements of Section 3.4.1 regarding the suitability of PCTP for sending large size agent codes if they are completely used in all the locations the agent visits. Nevertheless, notice that there is a case in which this statement is not completely true. It is the case of 1 and 10 agents using all the agent code (check the third row of Table 4.6). The efficiency of the messaging service implemented in JADE decreases as the size of the messages which are sent increases [Cuc04, JJK06]. Therefore, in this specific case it is more efficient to send the agent within several small messages than within a big one. When one hundred agents are migrating, this is different because the number of messages involved is so high that the messaging service gets rather saturated, and the average migration time raises.

According to the result tables, agents using only part of their code take advantage of an on-demand migration strategy, since less time is spent than using the push migration strategy. As just the two main classes of the agent are requested, only four messages are exchanged. The bigger the agent code is, the better is the improvement. Notice that the average time spent for each agent in this case is practically constant (in Figure 4.4 there are represented the different times spent for ten agents as the number of classes increase). The results, therefore, confirm the statements described in Section 3.4.2.

4.4 Performance evaluation 2: PCTP vs FrTP

This second part of the performance evaluation comprises the PCTP and the FrTP migration protocols. The aim of this evaluation is comparing the performance of the two mentioned protocols, and establishing which are the most appropriate FrTP fragment sizes for each situation. Since FrTP is devised to optimise the transference of weight agent resources (agent code and data), the tests have been performed with agents of different code sizes and, in some cases, of different data sizes (from 5KB to 1000KB). Each test has been done with 1 and 10 agents running simultaneously. The first case allows to analyse the time spent by a single agent migration round-trip. And the second case allows to analyse the average time consumed in concurrent migrations by each agent round-trip to the AM. All these tests have been performed in three different scenarios, since FrTP is devised for any type of network environment.

In the next tests these are the parameters modified: the size of the agent code and the agent data, the number of iterations or round-trips (which is adjusted to 10 for agents greater than 100KB, and to 100 for agents smaller or equal than 100KB), the number of agent instances concurrently running, and the FrTP fragment size. It must be taken into account that in all the cases the agent code is encapsulated in uncompressed JAR files.

4.4.1 Scenario 1: Local Area Network

The first scenario considered is a Local Area Network (LAN). This is the network explained in Section 4.2. It has an associated response time of less than 1ms, it does not present packet loss, and it has a performance of 100 Mb/s.

Two basic sets of tests have been performed in this scenario. The first one is a set of agents with different code sizes (see Table 4.7). The second one is a set of agents with different data sizes in the same environment than the previous one (see Table 4.8).

Firstly, as it is depicted in Figure 4.9 PCTP performs better than FrTP for small agent code sizes up to 25KB. This is because this protocol is less complex than the other, and due to the less number of messages used when the information to transfer is relatively small. But for agents with codes equal or higher than 50KB FrTP performs very much better than PCTP. The reason is that no weighty ACL messages are used and,

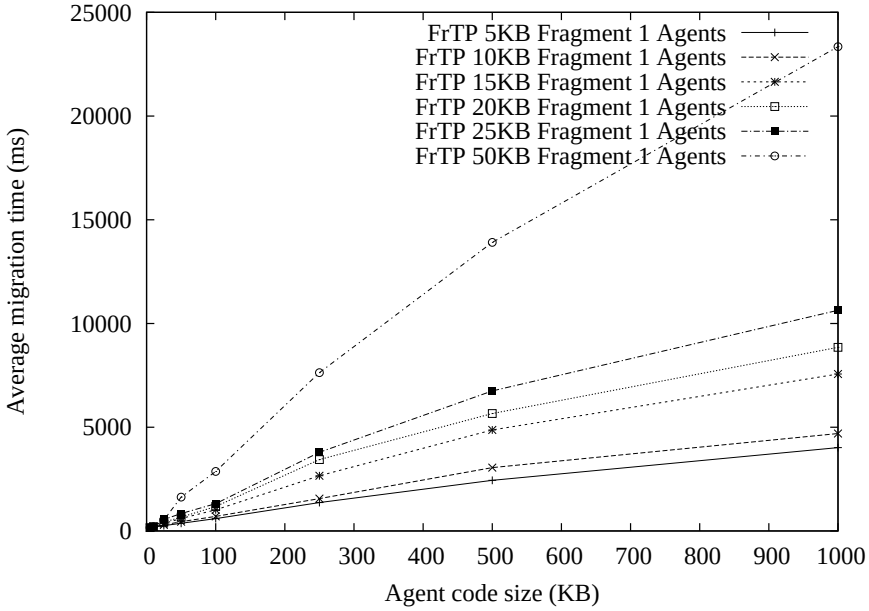

Figure 4.5: FrTP 1 Ag. Sc. 1 (code).

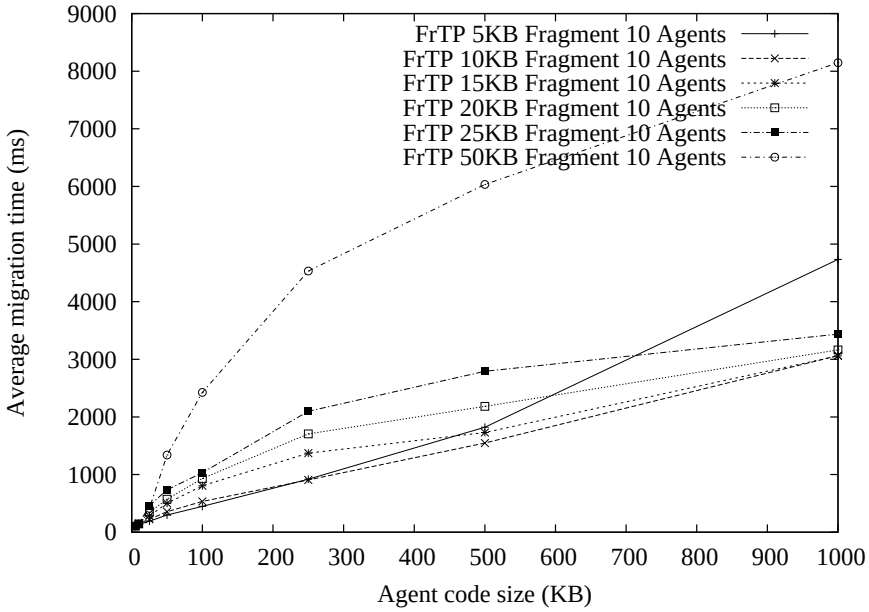

Figure 4.6: FrTP 10 Ag. Sc. 1 (code).

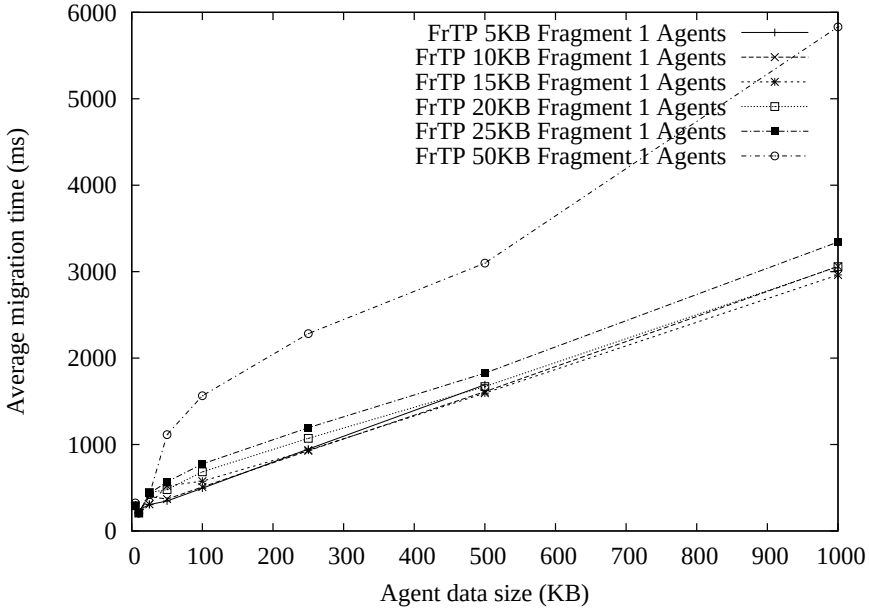

Figure 4.7: FrTP 1 Ag. Sc. 1 (data).

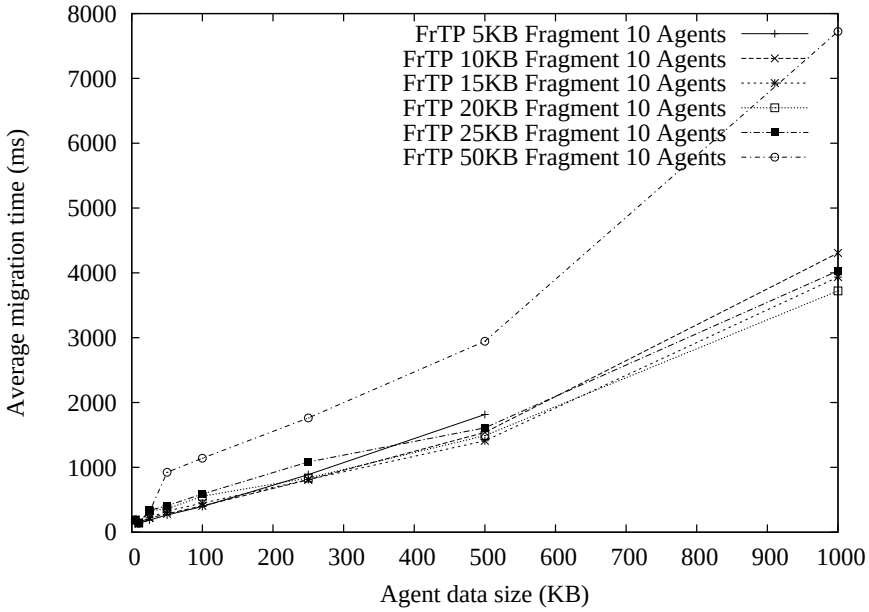

Figure 4.8: FrTP 10 Ag. Sc. 1 (data).

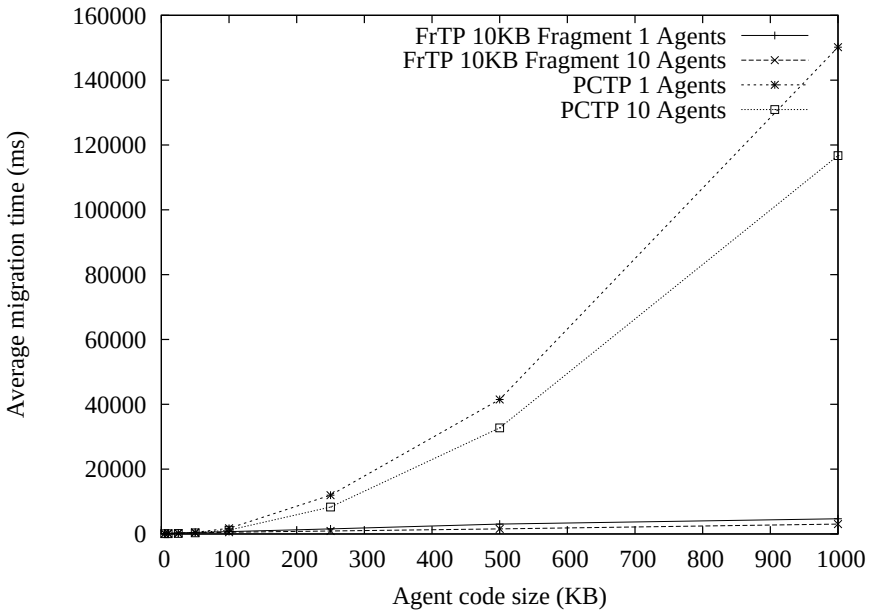

Figure 4.9: FrTP vs PCTP Scenario 1 (code).

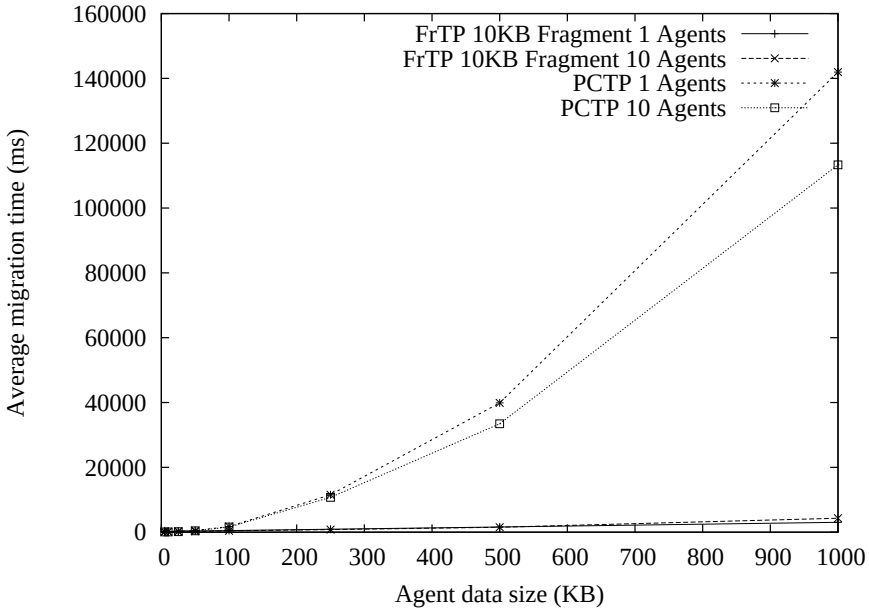

Figure 4.10: FrTP vs PCTP Scenario 1 (data).

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
FrTP	1	165	189	253	368	593	1,367	2,442	4,016
5 KB	10	112	132	190	297	448	919	1,822	4,733
FrTP	1	161	204	290	444	705	1,559	3,055	4,698
10KB	10	107	150	223	356	534	906	1,548	3,065
FrTP	1	161	205	359	600	1,020	2,658	4,871	7,563
15KB	10	107	142	283	502	805	1,372	1,729	3,061
FrTP	1	161	206	424	668	1,179	3,439	5,662	8,852
20KB	10	106	143	346	572	932	1,705	2,183	3,162
FrTP	1	161	204	551	838	1,314	3,789	6,750	10,638
25KB	10	107	142	460	735	1,033	2,098	2,792	3,438
FrTP	1	161	204	530	1,629	2,868	7,632	13,908	23,343
50KB	10	107	142	414	1,339	2,426	4,532	6,034	8,147
PCTP	1	126	140	222	493	1,734	11,943	41,499	150,162
	10	77	85	140	339	1,200	8,300	32,710	116,721

Table 4.7: Scenario 1: Multi-size agent code migration performance (in ms).

therefore, its associated overhead does not appear (see [Cuc04, JJK06]). According to the results, it is easy to see that when PCTP is used, the time spent in the migration increases exponentially as the agent code gets weightier. While it increases linearly when FrTP is used.

Regarding the migration concurrence, the two protocols perform better when several migrations are done in parallel. Nevertheless, FrTP seems to be specially favoured with it. The reason is that the transference is divided in many messages than can be interleaved with the messages of the other migrations. This is specially noticeable when large agent codes or data are transfered, where the time consumed with concurrence can be less than a 50% with a single agent.

In these tests a metric to compare the throughput of the migration transfer protocols is introduced. It consists of calculating the best transfer data rate achieved with each protocol. it is calculated by dividing the amount of data transfered (in our case twice of the agent code size, since there are two migrations in a migration round-trip) by the time spent. Then, the best throughput of FrTP is achieved by agents of 1000KB sent in fragments of 15KB. The throughput in this case is about 666KB/s. In the case of PCTP the better throughput is for agent codes of 25KB, and it is about 313KB/s. In case of using this protocol for agents of 1000KB, the throughput is only 17KB/s.

In FrTP (Figures 4.5 and 4.6) the best fragment sizes are between 5KB and 15KB

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
FrTP	1	300	243	305	347	495	947	1,692	-
5 KB	10	128	138	186	264	397	887	1,815	-
FrTP	1	287	209	398	369	509	928	1,611	3,062
10KB	10	192	149	217	284	405	810	1,539	4,307
FrTP	1	288	206	308	521	576	935	1,594	2,962
15KB	10	185	138	222	326	448	811	1,409	3,936
FrTP	1	292	207	445	478	684	1,071	1,670	3,057
20KB	10	186	140	331	360	555	836	1,489	3,722
FrTP	1	292	208	434	567	776	1,194	1,826	3,343
25KB	10	192	139	337	408	588	1,083	1,609	4,031
FrTP	1	324	206	420	1,114	1,564	2,283	3,099	5,832
50KB	10	192	139	310	923	1,141	1,762	2,944	7,724
PCTP	1	133	148	227	499	1,760	11,546	39,869	141,939
	10	81	91	160	413	1614	10,726	33,456	113,339

Table 4.8: Scenario 1: Multi-size agent data migration performance (in ms).

depending on the amount of data to transfer, and the number of agent migrations. When there is only one agent migration the fragment size of 5KB is the most appropriate. While, when there are ten agents migrating, because of the higher number of messages exchanged, the best fragment sizes are 5KB and 10KB depending on the amount of data to transfer (usually more data performs better with 10KB fragments).

Finally, since FrTP also optimises the transfer of the agent data, a set of tests with agents carrying different amounts of data have been performed (see Table 4.8). The results are similar to the previous ones, but they have some peculiarities because the data is created by means of an agent global variable of the desired size. First of all, the tests with 1000KB of data and fragments of 5KB cannot be performed because too much memory is used (our agent serialisation keeps in memory the agent instance and the serialisation result, this is at least the double amount of data in memory). Secondly, the time spent in agent migrations without concurrence is similar or even smaller than in agent migrations with concurrence. The reason is that the data to transfer is in memory, and it is more efficient to deal with it. Conversely, the agent code is kept into a secondary storage. Furthermore, the concurrence is penalised by the extra amount of memory used. And regarding the optimum fragment size, in this case it is more variable, and it ranges from 5KB to 15KB (Figures 4.7 and 4.8). PCTP performs similar to the previous cases with agents of different code sizes. In this case the difference between concurrent and

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
FrTP 10KB	1	2,300	2,768	3,628	5,230	8,551	18,531	35,203	68,362
	10	1,742	2,244	3,122	4,630	8,031	17,966	34,599	67,833
FrTP 15KB	1	2,298	2,565	3,372	4,824	7,550	15,903	30,061	58,032
	10	1,742	1,958	2,806	4,310	7,132	15,499	29,551	57,600
PCTP	1	2,017	2,264	2,719	3,423	5,216	16,496	46,232	153,689
	10	1,544	1,695	2,094	2,303	2,550	9,133	33,870	112,927

Table 4.9: Scenario 2A: Multi-size agent code migration performance (in ms).

no concurrent migrations have also been reduced.

4.4.2 Scenario 2: Wide Area Network

The second scenario considered is a Wide Area Network (WAN). The network, simulated with the NetEm [Hem05] Linux utility, has an associated response time of 120ms, with a variation of 10ms, and a bandwidth of 100 Mb/s. Two different cases have been considered, one where there is no packet loss (called Scenario 2A) and another one where there is a 5% packet loss (called Scenario 2B). In this last case, according to [MSM97], the maximum bandwidth is limited at the transport level by the TCP protocol because of the packet loss and retransmissions.

Only one set of tests for each of the network variants has been considered. The set of tests that analyses agent migrations with different code sizes (see Table 4.9 for Scenario 2A, and Table 4.10 for Scenario 2B). In this case only the 10KB and 15KB fragment sizes are tested, since the cost to perform the tests in these scenarios is higher than in the previous ones, and in this case the fragment size comparison is not so relevant. In the first scenario, the best fragment sizes were 5KB, 10KB, and 15KB. In this case only the 10KB and 15KB fragment sizes have been used, since they imply less messages sent (this is better with this scenario latency). At the end, the best results have been achieved by the 15KB fragment size (this is the reason why the 10KB results are not represented in the figures).

In the first case (Table 4.9, and Figure 4.11), the most relevant consequence of the network latency rise is an important increase of the average time spent by the agent migrations. The network latency does not affect the effective network bandwidth. But the migration transfer protocols presented are victims of the message handshake delays,

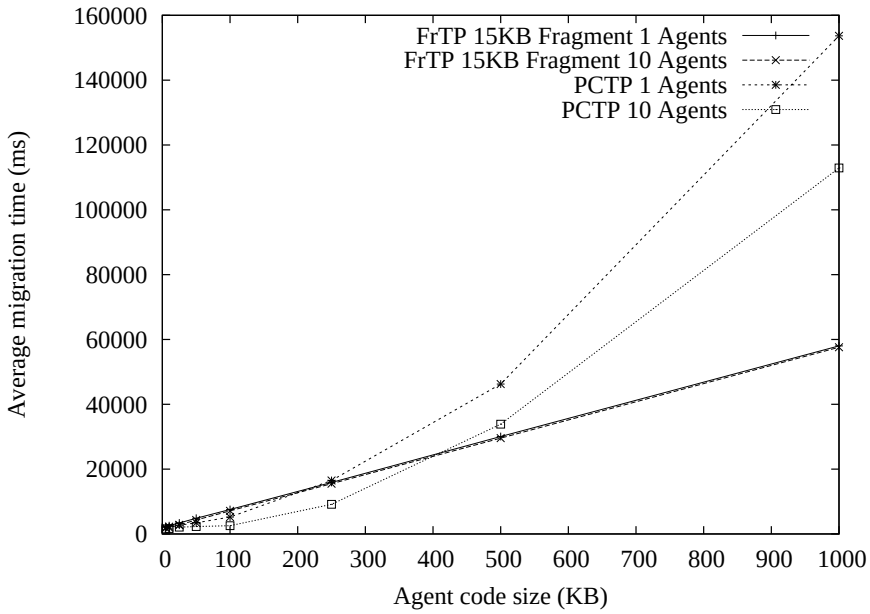

Figure 4.11: FrTP vs PCTP Scenario 2A.

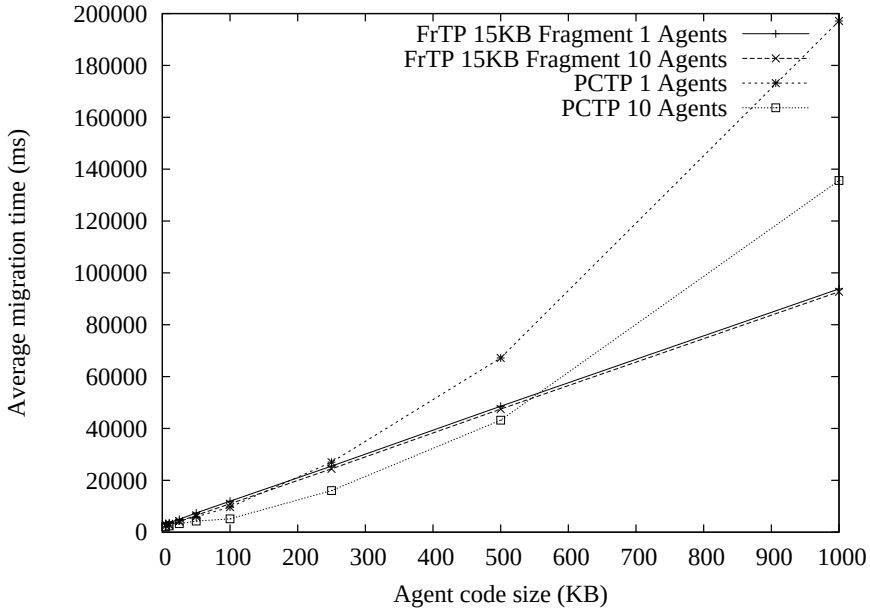

Figure 4.12: FrTP vs PCTP Scenario 2B.

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
FrTP 10KB	1	3,285	3,959	5,333	7,817	13,065	28,031	53,966	106,269
	10	2,456	3,104	4,463	6,875	12,071	27,670	53,423	10,5216
FrTP 15KB	1	3,359	3,665	4,962	7,447	11,957	25,457	48,561	93,828
	10	2,462	2,738	4,129	6,515	10,942	24,426	47,469	92,693
PCTP	1	2,988	3,248	4,276	5,932	9,673	27,001	67,196	197,131
	10	2,189	2,424	3,289	4,284	5,123	16,034	43,207	135,615

Table 4.10: Scenario 2B: Multi-size agent code migration performance (in ms).

highly increased because of the network latency. Nevertheless, it is surprising that the round-trip time increase is specially noticeable with FrTP. On the one hand, it seems to make sense that as more messages are sent more noticeable is the latency, but on the other hand, FrTP is designed to mitigate this problem, since no acknowledgements are expected by each data message sent. Then, the explanation is that despite the protocol does not wait for acknowledgements, the underlying transport protocol, which is implemented by the MTP-HTTP, does. Therefore, the delay in this case is directly proportional to the number of fragments sent. Using an MTP that did not require an acknowledge for each message sent, such as the MTP-UDP [Cuc04], the performance of FrTP, in such conditions, would be rather better. Regarding PCTP, although the current network conditions also increase the average round-trip time, in migrations of large agents the latency is concealed by the penalisation of ACL messages with high amounts of data. Furthermore, it must be taken into account that in this case the number of exchanged messages is much smaller than in FrTP.

Comparing the two protocols, in this scenario, FrTP performs better than PCTP for agent codes equal or greater than 500KB. The best transfer data rate achieved with FrTP is 35KB/s with the 1000KB agent and 15KB fragments. At the contrary, the best transfer data rate achieved with PCTP is 78KB/s with the 100KB agent. The same protocol with the 1000KB agent presents a transfer data rate of 18KB/s. All the cases are referred to concurrent migrations.

In the second case (Table 4.10, and Figure 4.12) the results obtained are conceptually similar to the previous ones. The only difference is a general increase of the average time spent for each migration round-trip (approximately a 50% more than the

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
FrTP 10KB	1	1,169	1,524	2,001	2,859	4,695	10,181	19,327	37,589
	10	935	1,149	1,652	2,499	4,329	9,795	18,870	37,053
FrTP 15KB	1	1,177	1,425	1,975	2,799	4,474	9,736	18,159	35,217
	10	935	1,035	1,562	2,465	4,129	9,212	17,684	34,617
PCTP	1	1,008	1,100	1,675	2,496	5,002	19,412	56,628	177,940
	10	801	882	1,253	1,493	2,714	12,205	39,581	125,591

Table 4.11: Scenario 3: Multi-size agent code migration performance (in ms).

results obtained in the previous scenario). The transfer data rate achieved in this scenario with FrTP is 21KB/s with the 1000KB agent and 15KB fragments. Using PCTP the maximum data rate achieved is 39KB/s with the 100KB agent. The same protocol with the 1000KB agent presents a transfer data rate of 15KB/s. All the cases are referred to concurrent migrations. Finally, it must be taken into account that in this scenario the network bandwidth is limited because of the latency and packet loss (see [MSM97]). Nevertheless, this limit does not affect the mentioned data transferences because the data rate obtained is below it.

4.4.3 Scenario 3: Metropolitan Area Network

The third scenario considered is a Metropolitan Area Network (MAN). The network, simulated with the NetEm [Hem05] utility, is composed of two 20Mb/s (downstream) 1Mb/s (upstream) ADSL links to a Digital Subscriber Line Access Multiplexer (DSLAM) that behaves as a network hub. The response time associated to these links is around 30ms. The data exchanged between Agent Platforms (APs), since traverses two ADSL links, can only reach 1Mb/s of bandwidth and the response time achieved is 60ms.

As in the previous scenario, only one set of tests has been considered. The set of tests that analyses agent migrations with different code sizes (see Table 4.11). In this case the average time spent in an agent migration round-trip has been proportionally reduced to the latency reduction regarding the Scenario 2A. In this case, as it is shown in Figure 4.13, FrTP performs better than PCTP for agents equal or greater than 100KB, when there is a single agent migrating, and 250KB, for ten agents migrating. The transfer data rate achieved in this scenario with FrTP is 58KB/s with the 1000KB agent and 15KB fragments. Using PCTP the maximum data rate achieved is 74KB/s with the

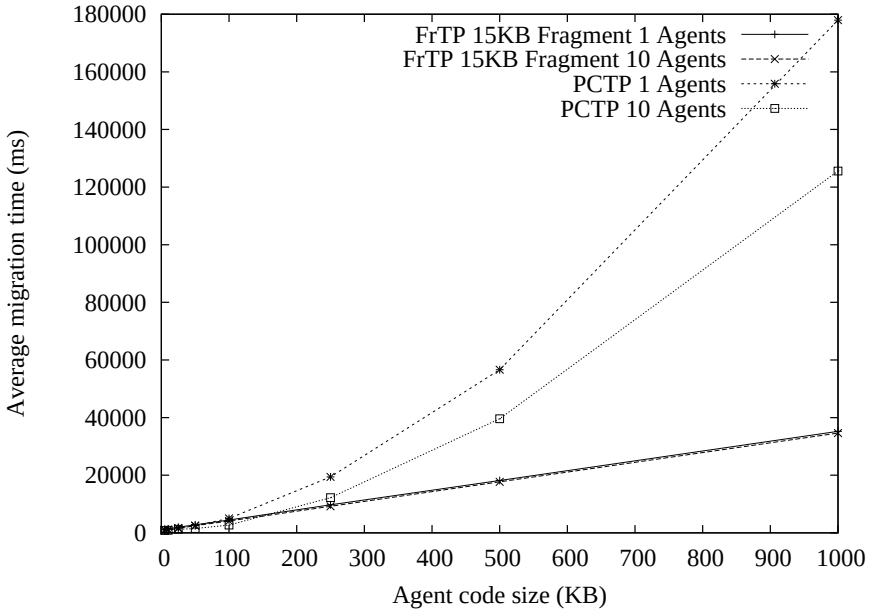

Figure 4.13: FrTP vs PCTP Scenario 3.

100KB agent. The same protocol with the 1000KB agent presents a transfer data rate of 16KB/s. All the cases are referred to concurrent migrations. It is worth noticing that, despite the bandwidth limitation of the links present in this scenario, the maximum data rate achieved does not reach a 100% usage of the link. The reason is that the effective data rate achievable by the two protocols is limited by the latency in FrTP, and the poor efficiency of the Agent Communication Language (ACL) messages in JADE when they contain high amounts of data, such as in PCTP.

4.5 Performance evaluation 3: PCTP vs RESTTP

This third part of the performance evaluation comprises the PCTP and the RESTTP migration protocols. The aim of this evaluation is comparing the performance of both protocols. Since RESTTP, as FrTP, is devised to optimise the transference of weight agent resources (agent code and data), the tests have been performed with agents of different code sizes and, in some cases, of different data sizes (usually from 5KB to 1000KB). Each test has been done, with 1, and 10 agents running simultaneously. All

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	101	102	106	109	126	160	212	311
	10	71	71	73	77	87	108	145	242
PCTP	1	126	140	222	493	1,734	11,943	41,499	150,162
	10	77	85	140	339	1,200	8,300	32,710	116,721

Table 4.12: Scenario 1: Multi-size agent code migration performance (in ms).

these tests have been performed in the same three scenarios used in the previous section, since RESTTP is also devised for any type of network environment.

In the next tests these are the parameters modified: the size of the agent code and the agent data, the number of iterations or round-trips (which is adjusted to 10 for agents greater than 100KB, and to 100 for agents smaller or equal than 100KB), and the number of agent instances concurrently running. In all the cases the agent code is also encapsulated in uncompressed JAR files.

4.5.1 Scenario 1: Local Area Network

Two basic sets of tests have been performed in this scenario. The first is a set of agents with different code sizes, migrating between the two locations present in the network, and using RESTTP and PCTP (Table 4.12). The second is a set of agents with different data sizes in the same environment than the previous one (see Table 4.13).

Firstly, as it is depicted in Figure 4.14, RESTTP is the most efficient protocol. The average migration round-trip time increases linearly, while using PCTP this time increases exponentially as agents get weightier. PCTP can be considered usable to migrate agents with a code size up to 50KB with regard to RESTTP. The migration concurrency reduces the average migration time spent by the AM to a 66% of the time spent for a single migrating agent.

The best data transference rate obtained by RESTTP is achieved by agents of 1000KB. The throughput in this case if about 8MB/s. In the case of PCTP the better throughput is for agent codes of 25KB, and it is about 313KB/s. In case of using this protocol for agents of 1000KB, the throughput is only 17KB/s.

Finally, since RESTTP also optimises the transfer of the agent data, a set of tests with agents carrying different amounts of data has been performed (Table 4.13 and

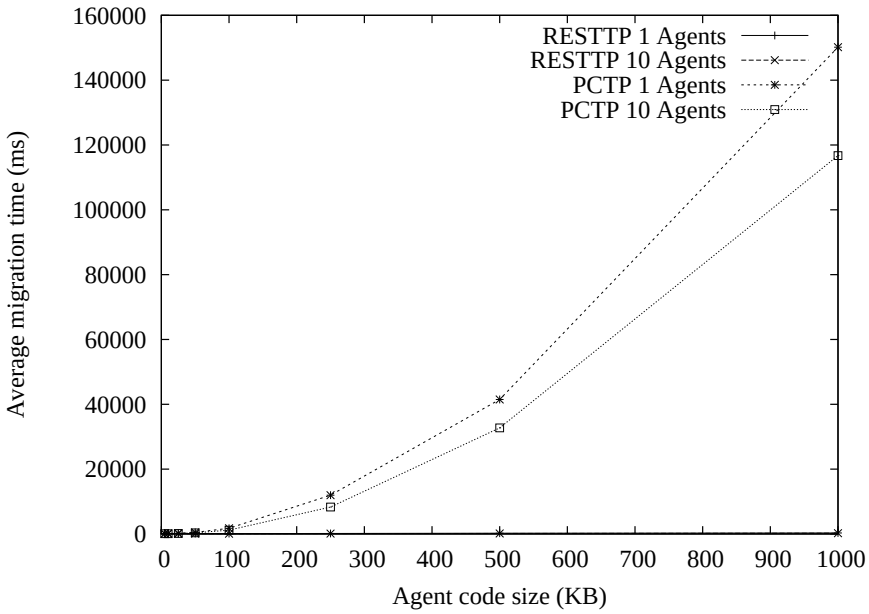

Figure 4.14: RESTTP vs PCTP Scenario 1 (code).

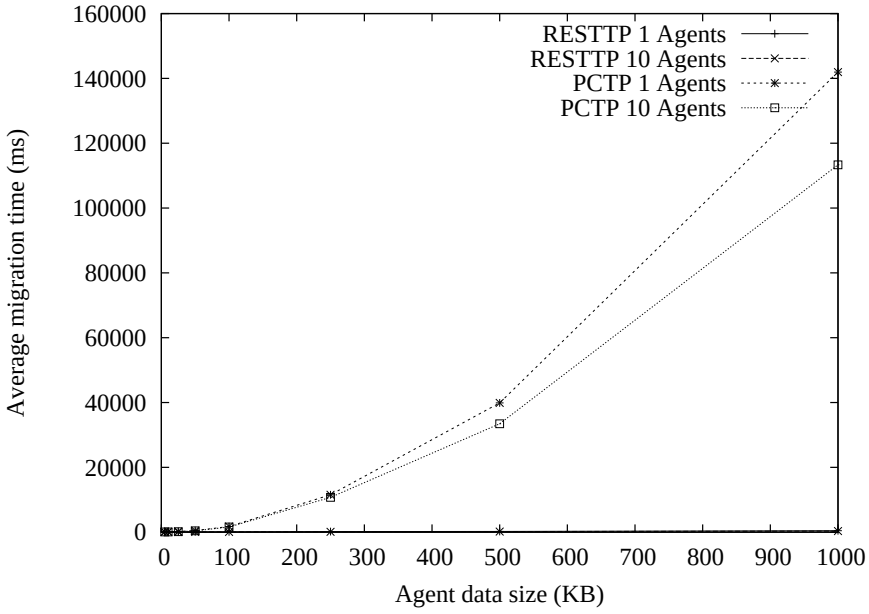

Figure 4.15: RESTTP vs PCTP Scenario 1 (data).

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	102	103	106	112	130	174	240	444
	10	71	72	74	79	91	123	186	330
PCTP	1	133	148	227	499	1,760	11,546	39,869	141,939
	10	81	91	160	413	1,614	10,726	33,456	113,339

Table 4.13: Scenario 1: Multi-size agent data migration performance (in ms).

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	2,502	2,727	3,027	3,451	3,998	5,037	6,128	7,868
	10	1,315	1,365	1,527	1,758	3,590	4,743	5,822	7,558
PCTP	1	2,017	2,264	2,719	3,423	5,216	16,496	46,232	153,689
	10	1,544	1,695	2,094	2,303	2,550	9,133	33,870	112,927

Table 4.14: Scenario 2A: Multi-size agent code migration performance (in ms).

Figure 4.15). The results are quite similar to the previous ones with only slight changes. They are consequence of the different treatment that the agent data receives regarding the agent code.

4.5.2 Scenario 2: Wide Area Network

As in the previous section, only one set of tests for each of the network variants (the one without packet loss and the one with it) has been considered. The set of tests that analyses migrations with different agent code sizes (see Table 4.14 for Scenario 2A, and Table 4.15 for Scenario 2B).

In the first case (Table 4.14 and Figure 4.16), the network latency strongly penalises protocol handshakes. Since not only does RESTTP exchange the same number of ACL messages than PCTP, but it even establishes two HTTP connections, RESTTP performs worse than PCTP when a single agent with a code size between 5KB and 50KB migrates. The time spent in the number of operations required to transfer such amounts of data is too high to be recovered. In all the other cases RESTTP performs better than PCTP. Furthermore, the migration concurrency also improves the average performance results.

The best transfer data rate achieved with RESTTP is 265KB/s with the 1000KB agent. At the contrary, the best transfer data rate achieved with PCTP is 78KB/s with the 100KB agent. The same protocol with the 1000KB agent presents a transfer data

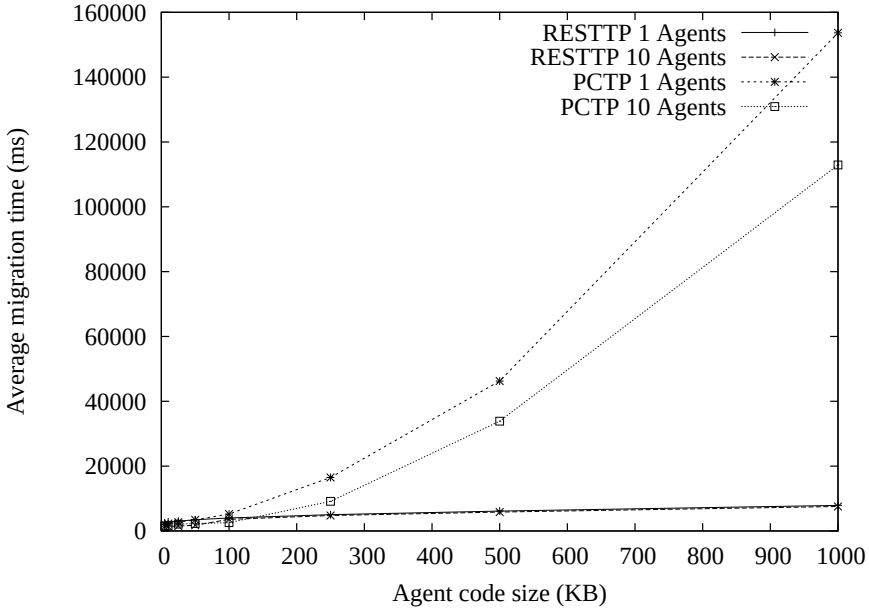

Figure 4.16: RESTTP vs PCTP Scenario 2A.

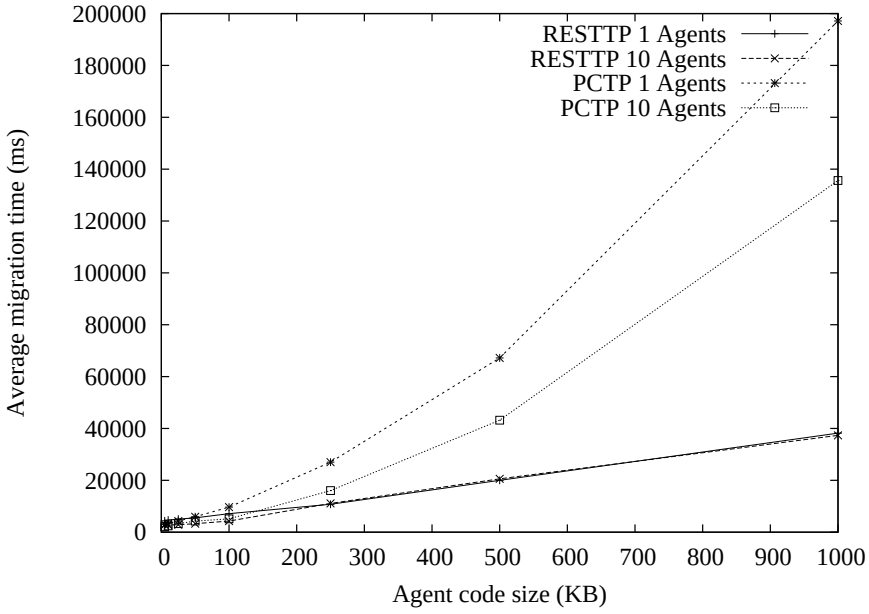

Figure 4.17: RESTTP vs PCTP Scenario 2B.

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	4,382	4,656	5,033	5,505	7,160	10,760	20,019	38,209
	10	2,707	2,746	2,844	3,269	4,275	11,114	20,548	37,354
PCTP	1	2,988	3,248	4,276	5,932	9,673	27,001	67,196	197,131
	10	2,189	2,424	3,289	4,284	5,123	16,034	43,207	135,615

Table 4.15: Scenario 2B: Multi-size agent code migration performance (in ms).

Protocol	Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	1,357	1,443	1,683	2,131	2,972	5,670	10,134	18,998
	10	701	742	1,012	1,224	2,271	5,532	9,995	18,868
PCTP	1	1,008	1,100	1,675	2,496	5,002	19,412	56,628	177,940
	10	801	882	1,253	1,493	2,714	12,205	39,581	125,591

Table 4.16: Scenario 3: Multi-size agent code migration performance (in ms).

rate of 18KB/s. All the cases are referred to concurrent migrations.

In the second case (Table 4.15, and Figure 4.17) the combination of latency and packet loss increases the packet exchange penalisation, and the channel bandwidth. In this case RESTTP performs better than PCTP with agent code sizes from 50KB without concurrent migrations, and from 25KB with concurrent migrations. Migration concurrency reduces the latency effects, but only in the ACL message exchange (the improvement obtained because of the parallelisation is higher in small agent codes).

The transfer data rate achieved in this scenario with RESTTP is 54KB/s with the 1000KB agent. Using PCTP the maximum data rate achieved is 39KB/s with the 100KB agent. The same protocol with the 1000KB agent presents a transfer data rate of 15KB/s. All the cases are referred to concurrent migrations. In this case the maximum data rate obtained is not only limited by the latency, but by the combination with the message loss that, under the TCP protocol, establishes a maximum channel bandwidth. This can be proved checking the performance results obtained by agents with large agent codes sent with RESTTP. In these cases the time spent increases proportionally and linearly with regard to the code size (see first and second rows of Table 4.15), which means a bandwidth limitation. In other cases, the time spent does not increase proportionally (see first and second rows of Table 4.14).

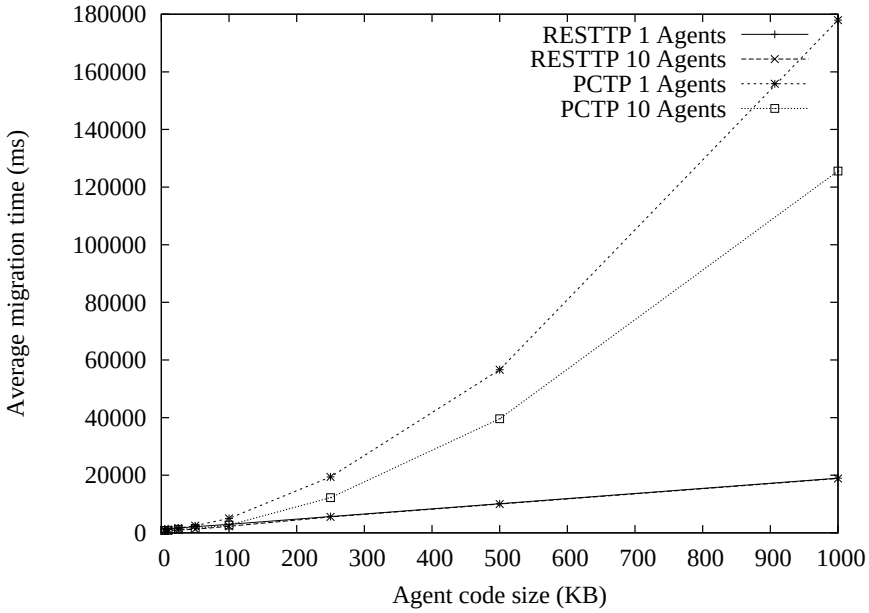

Figure 4.18: RESTTP vs PCTP Scenario 3.

4.5.3 Scenario 3: Metropolitan Area Network

As in the previous scenario, only one set of tests have been considered. The set of tests that analyses agent migrations with different code sizes (Table 4.16 and Figure 4.18). In this case RESTTP is the protocol with the best performance, with the exception of single migrating agents with code sizes between 5KB and 25KB.

The transfer data rate achieved in this scenario with RESTTP is 106KB/s with the 1000KB agent. Using PCTP the maximum data rate achieved is 74KB/s with the 100KB agent. The same protocol with the 1000KB agent presents a transfer data rate of 16KB/s. All the cases are referred to concurrent migrations. In this case RESTTP use the 100% of the channel bandwidth (since it is 1Mb/s and the transfer reaches 106KB/s).

4.6 Conclusions

In this chapter, through the use of the JIPMS implementation, it has been demonstrated the versatility of IPMA. Different migration transfer protocols give rise to different migration strategies suitable to one or more specific environments. And each of these

protocols obtain different results depending on the environment conditions. In the next paragraphs there is a comparison of the benefits of each one taking into account the results previously obtained.

The protocols analysed have been PCTP, ODTP, FrTP, and RESTTP. The first three are only based on the use of IEEE-FIPA standards, whereas the last one introduces the use of HTTP requests. PCTP, and FrTP are based on the push migration strategy, RESTTP is based on a pull-at-once migration strategy, and ODTP in a pull-on-demand migration strategy (see Section 3.4 for more details).

PCTP is the default protocol of JIPMS. It is the simplest one. This is the reason why it has been chosen as the reference protocol for the comparisons. According to the results obtained, this protocol is suitable for migrating agents with a small agent code and small agent data (typically up to 50KB). Data transferences of more than 250KB are usually highly inefficient because of the ACL unsuitability for dealing with large amounts of data (see [Cuc04, JJK06]). Nevertheless, when there are network performance restrictions (high latency, packet loss, or bandwidth limitation) it may be preferable its usage even with larger agent codes and data.

FrTP is a protocol that sends the agent code and agent data spread in several fragments, with a customisable size, each one encapsulated in an ACL message. This takes advantage of the JADE messaging service suitability for dealing with huge amounts of small ACL messages [CGK+05]. It was initially devised to substitute PCTP, since by changing the fragment size it can be generalised to the case of PCTP, where all the agent code is sent in a single ACL message, and all the agent data is sent in another one. Nevertheless, according to the results, it is better to use it as a complement of PCTP, since the complexity increase of FrTP is too high for small agents (5KB to 25KB agent codes), where PCTP performs better. For large agent codes the performance is quite better. There is another disadvantage, in case of network latency the protocol is not so efficient as initially anticipated, because of the high amount of messages which are sent. Despite the fact that there is no acknowledgement of these messages, the MTP-HTTP does not consider a message sent until an internal HTTP response is received. This is the reason why the latency affects the performance of the protocol. Using another MTP without this restriction the performance would be better in this case.

RESTTP is a protocol based on the request of the agent resources, from the remote location, using HTTP connections. This protocol demonstrates that the data transferences based on ACL messages are not efficient. In most of the cases it takes advantage of almost 100% of the bandwidth available. This protocol is suitable for almost all the environments. The only disadvantages it has are the use of additional standards than the ones dictated by the IEEE-FIPA, and that it requires establishing new connections from the remote platform to the local platform (sometimes this may be a problem because new ports must be opened in network firewalls).

And ODTP is a protocol that only transfers the required agent code as it is needed. It is devised for agents which do not use all their code in all the locations they visit. It is intended to be used only in local area networks, since the cost of individually requesting several code snippets can be very high in wide area networks where the associated latency is higher. Checking the performance results, it can be seen that when only few parts of the agent code are used, the time spent in the migrations is constant. Then, when large agent codes are involved, and they are in the appropriate environment, this protocol can even be more efficient than FrTP.

Finally, the better strategy to migrate agents, if only IEEE-FIPA standards can be used, is the combination of several protocols. Each one used in the most suitable case for it. In case of being able to use non IEEE-FIPA standards, then the response is clear, RESTTP is the most efficient protocol in almost all the situations. Nevertheless, it must be said that all the given results depend on the specific implementation tested, and that they could present some differences if other implementations and MTPs were used.

Chapter 5

Agent Code Distribution Service

This chapter proposes a global cache service to efficiently and securely deal with the distribution of agent code. An implementation of the service is presented, and a set of performance tests demonstrate its benefits.

5.1 Introduction

The performance of mobile agent migrations has been always penalised because of the need of carrying the three parts from which agents are composed (code, data, and state) to each visited location (see Section 2.2.2). The agent data and state must always be transfered from location to location since they dynamically change. Nevertheless, since the agent code is static during the whole life of the agent, its management can be improved, e.g., using code caches, and the penalisation associated to its transmission can be reduced [Gav04]. Furthermore, in case several agents share the same code, a huge quantity of network bandwidth can be saved up. Therefore, an intelligent management of the agent code can improve the migration time of agents.

Several solutions (see Chapter 3 or [BR05]), such as the addition of code caches or the partial transference of the agent code (code on demand), have been used in Agent Platforms (APs) to improve the agent performance. Nevertheless, these solutions are always local to APs. Agent codes can be better managed from a global point of view, taking into account the code necessities everywhere.

In this chapter a service called Agent Code Distribution Service (ACDS) is proposed to manage the agent code from a global perspective. In Section 5.2 a set of requirements for a global code manager service is detailed. Then in Section 5.3, ACDS with its security features and its public user interface is explained. The service has been implemented allowing the performance of several tests in different scenarios to demonstrate its benefits, see Section 5.4. Then, a set of related work is compared with the service proposed in Section 5.5. And, finally, Section 5.6 concludes the chapter.

5.2 Requirements and Roles

Regarding the nature of the service which is proposed later in this chapter, a set of requirements have been selected in the following lines. They are taken into account in its design, which is explained in the next section.

- *Agent middleware independence.* The codes managed by the service and the interaction of it with APs must not be limited to a specific type of Agent Middleware (AM).

- *Programming language independence.* Both, the interface to access the service and the codes managed by it must not be restricted to a specific Programming Language (PL).

- *Code efficiently transported.* Agent code must be transfered using efficient protocols. The performance should be independent of the amount of data transmitted. E.g., in the JADE [BCPR08] AM the ACL messaging system performance decrease as the ACL messages size grows [Cuc04, JJK06].

- *Code intelligently distributed and cached.* Agent code must be distributed and cached close to the APs that possibly will need it in a nearby future.

- *Code distribution under contract.* The parameters regarding the distribution and maintenance of code must be associated to a code contract.

- *Support for multiple code binaries.* Each agent code must be able to include one or more binaries developed in different PLs. This permits the support for inter-language agents (see Chapter 6).

- *Support for code updates.* Agent code updates must be able to be propagated for security purposes.

- *Secure code distribution.* Sensitive operations, such as removing or updating an existing agent code, must always be authenticated. Furthermore, the agent code integrity and authenticity must be guaranteed.

- *Service transparent to agents.* The fact of using a service to manage the agent code must be transparent to agents, which are not needed to be specifically developed to support this situation.

But not only the service requirements must be taken into account for its design. Any service has at least two parties involved in it, the client and the server. In this case, there are more parties that take part in it. Each one is represented with a specific role:

- *Code developer*: This is the role associated to the agent developer. It has nothing to do with the code distribution service since the service is transparent to the agent.

- *Code owner*: This is the role associated to the person who has bought, or sometimes developed, the agent. It uses the agent code service to voluntarily or involuntarily distribute its code.

- *Code user*: This is the role associated to the agent on itself and to the AMs that use the code to run the agent.

- *Code provider*: This is the role associated to the code distribution service administrators.

All these roles are taken into account in the design of the service presented in the next section.

5.3 Agent Code Distribution Service

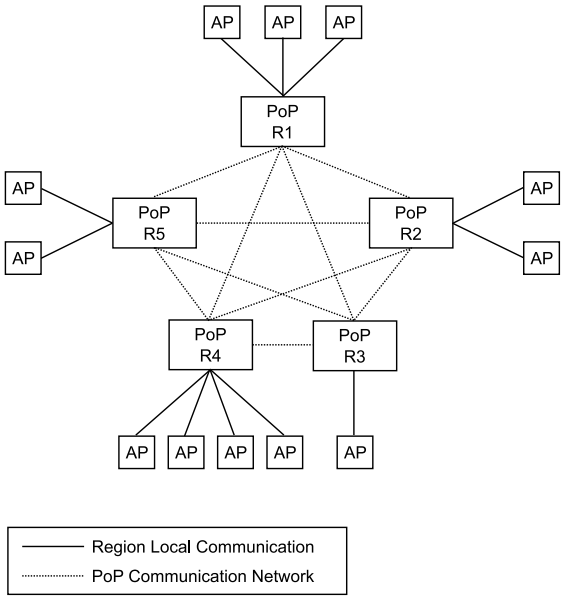

Figure 5.1: Agent Code Distribution System.

In this section a service to manage the distribution of agent codes, the ACDS, is presented. The agent code distribution is tackled from a global point of view and fulfils the requirements presented in the previous section.

5.3.1 Overview of the Architecture

ACDS defines a network composed of several nodes, called Point of Presences (PoPs), which distribute and cache code through inter-networks (see Figure 5.1). The idea is similar to the Content Delivery Networks (CDNs) [PB07], such as Akamai [DMP+02], but focused on agent codes, where critical information, from the performance point of view, is transported using a dedicated network belonging to the code provider.

The service is offered over an inter-network which is divided into regions (see Figure 5.1). Each region has one or more nearby networks assigned. There is a publicly known PoP in each region in addition to other ones for fault tolerance or load distribution purposes. PoPs are interconnected with themselves, compose the core of the

system, and are the smallest independent entities of ACDS. APs interact with their closest PoP for requesting agent codes. As closer is a PoP of an AP, better the performance that is got in the code fetching. As shown in Figure 5.2, a PoP node is composed of four basic components: the Local Code Repository (LCR), the Remote Code Manager (RCM), the Contract Enforcement Module (CEM), and the ACDS Interface.

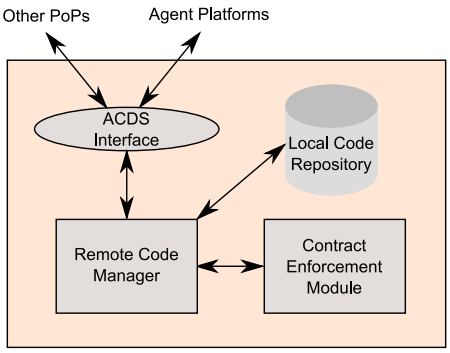

Figure 5.2: Point of Presence (PoP).

- *Local Code Repository (LCR):* It is a repository that contains all the agent codes local to a PoP. Non present codes in this repository must be requested to other PoP nodes.

- *Remote Code Manager (RCM):* It is the core of the PoP. It manages the local and remote operations to agent codes, i.e., it serves codes to APs, and requests and serves codes to other nodes, through the ACDS Interface.

- *Contract Enforcement Module (CEM):* This module enforces the rules specified in the contract of each agent code. Some operations will not be permitted if they are not authorised by the code contract.

- *ACDS Interface:* This is an interface between PoPs and clients, and PoPs and other PoPs. Two basic kinds of operations are supported, the request of an agent code, carried out by APs or other PoPs, and the management of published codes, carried out by the code owners or other PoPs. This interface is specified in Section 5.3.5.

Group Code Identifier (CGID): *1f332e209d0dc354*		
Code Identifier (CID)	**Security Revision (SR)**	**Hash Code Identifier (HCID)**
22d1ccfe8a	*3*	*ebd6c336cd3cebf83f5410a180112d41*
cb3acb26ce	*1*	*f368f50481a99b0b44b083d24e5b03d6*
27de3d6279	*2*	*225ea33d20e0185d1f4a5aa271e482a4*
56a7c9daa8	*1*	*2a51345179ead3700c8b2fecd150e222*
27de3d6279	*1*	*aff05f9767ce0dbb4e5bed7ae67518fe*

Figure 5.3: Example of the use of agent code identifiers.

5.3.2 Code Management

As previously mentioned, ACDS is a system similar to a CDN. Nevertheless, dealing with distribution and caching of code is slightly different from dealing with typical web data. Agent codes are identified by a different mechanism, sometimes they must be immediately updated or removed from the whole network, and different versions of their binaries might exist to support inter-language interoperability mechanisms such as the one proposed in Chapter 6, where an agent can be composed of several equivalent codes. Agent codes can be completely different regarding several aspects:

- the PL in which they are written,
- if they are compiled or are source code,
- the architecture or interpret for which they are prepared,
- the method of packaging

The ACDS can deal with any kind of codes, included source code packages, since the service does not interpret them. Codes are only interpreted by the AP which has requested them. The set of identifiers described in Section 3.3.2 is used to classify and uniquely identify codes (see also the example of Figure 5.3). Nevertheless, ACDS arranges them in an URN-like name, following the hierarchical schema shown in Figure 5.4, as:

urn:agent-code-id:<CGID>:<CID>:<SR>:<HCID>

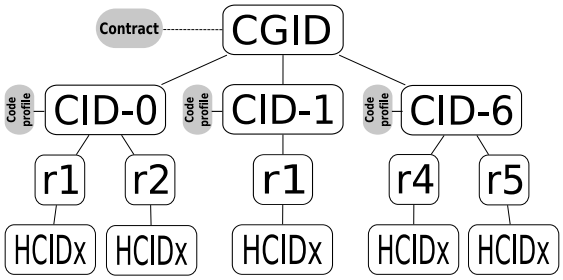

Figure 5.4: Code identifier hierarchy.

So, for example a given code can be identified by a name like:

urn:agent-code-id:cgid:cid-0:2:HCIDx.

This code is named following the identifier hierarchy shown in Figure 5.4, so regarding the code examples of Figure 5.3, the identifier:

urn:agent-code-id:1f332e209d0dc354

denotes all the code with $CGID = 1f332e209d0dc354$. Or the name

urn:agent-code-id:1f332e209d0dc354:22d1ccfe8a

denotes all the code revisions of the code with the $CID = 22d1ccfe8a$ and $CGID = 1f332e209d0dc354$.

Furthermore, each agent code must have associated a *code profile* in addition to the URN previously proposed. No specific format is enforced for the code profile, since the service only propagates it together with the code without requiring its interpretation. The code profile contains information, among others, of the PL in which the code is written and the type of architecture supported. This allows agent developers to create new agent codes supporting different profiles for a specific agent (see Section 6.5.3) which is already deployed. Existing agents can migrate to previously unsupported locations finding the new codes developed by using their Code Group Identifier (CGID), and the code profile information available in ACDS. The CGID is associated to the agent and represents all the agent codes suitable for this agent. Furthermore, as it is explained in Section 5.3.4, the agent code also has associated a *code distribution contract*.

5.3.3 Code Distribution

Two types of code distribution are supported by ACDS, the on demand distribution and the a priori distribution. They are used depending on the type of service required and the availability of a specific code.

On Demand Distribution

In the on demand distribution the code is uploaded to a specific PoP when it is needed. This way to distribute the code is flexible, but it does not always represent an improvement in the code fetching. If the code is only requested one time in each region, from the agent point of view, this may increase the final time spent to fetch it. It also has the disadvantage that, once the code has been spread through several nodes, it is not trivial to delete or update it. Two types of service can be offered depending on the code provider policy.

- On the one hand, the ACDS only acts as a region cache. In this case, if a code is requested by an AP and it is not present in the PoP, a response indicating that the code is not found is returned. Then, it is responsibility of the AP to upload the code, for the other's region APs benefit, after the code is traditionally received.
- On the other hand, another more advanced type of service allows the PoP to search for a requested agent code in other PoPs. Although different alternatives can be used, we propose to take advantage of a Distributed Hash Table (DHT), e.g., Chord [SMLN$^+$03]. The idea is to store in a DHT the agent code contracts, which include a list of the regions where the code is available. Therefore, each time a code is required the contract associated to the code requested is retrieved from the DHT, and one PoP of the list of regions is asked for the code. If the contract is not found in the DHT a code not found response is returned, and, as in the previous case, it is responsibility of the AP to upload the code once it is got.

A Priori Distribution

In the a priori distribution the code is uploaded to a set of PoPs according to a *list of regions* stated in the code distribution contract. Although this method is not as flexible as

the previous alternative, it guarantees that the code is present in all the desired regions when agents reach them. Furthermore, it allows to easily update and delete the distributed code. In case the code owner agrees, this distribution method can be combined with the previous one.

The flexibility of this method has been enhanced by using the concept of neighbourhood. Each PoP includes a list of its neighbours (PoPs), which denotes the closest nodes from a topographic point of view, that is defined by the node administrators, and it is static, i.e., it does not frequently change. Therefore, when a code is sent to the regions listed in the contract, each PoP associated to the region sends the code to its neighbours. This operation is repeated a number of times, according to a parameter called *neighbourhood degree*, down the tree of neighbours. The parameter is an integer that can be:

- **Zero**: in this case the code is not sent to any neighbour, i.e., this is equivalent to the original distribution method.
- **Greater than zero**: in this case there is a wider code distribution, while keeping the possibility to update and delete the code from all the nodes, since the pathway to them can be easily reconstructed.

As the neighbourhood degree increases, these processes are more error-prone. Nevertheless, depending on the average number of neighbours each node has, the neighbourhood degree has a limit which has no sense to be exceeded (see the concept of *Six degrees of separation* in [Bar02]).

5.3.4 Security Management

In the business model proposed for ACDS, where the agent code is distributed a priori, the code owner establishes a contract with the ACDS administration authority in order to use the network to distribute its code. This contract allows the code owner to use some of the distribution schemes available in ACDS under the stated conditions and possible constraints. Thus, ACDS must guarantee that the code will be distributed as it has been agreed and, furthermore, the infrastructure must provide security mechanisms

to protect the code and the code owner operations. One of the biggest threats to ACDS is the malicious code upload, update, or deletion by unauthorised users, so each PoP must enforce the user contract and other possible security policies applicable.

On the other hand, once the code is distributed over the network the code user can freely download it from the ACDS PoPs. No restrictions are initially imposed in code downloads regarding the code user. If download restrictions must be applied, they are stated in the initial contract by the code owner by, for instance, limiting the number of code downloads or bandwidth used during a period of time. This also allows to improve the code download efficiency and performance, by speeding up the process, because it can avoid servers saturation.

The agent code and agent code profile integrity can be guaranteed by signatures issued by the code owner. The service does not check these signatures, only distribute them along the agent code and agent code profile. Therefore, they can be verified by the AP which has requested them.

Code Distribution Contract

The code distribution contract is established with the ACDS administrative authority and states mainly what the code owner can distribute over the ACDS network and how.

The main elements contained in the contract are:

- *Contract issuer*: the ACDS authority issuing the contract.
- *Subject*: the code owner allowed to distribute code (a hash of the code owner public key is used as user identification).
- *Actions*: actions allowed to the code owner. This will normally be a subset of $\{upload, update, delete\}$.
- *Object*: the code to be distributed by ACDS.
- *Conditions*: several conditions to be met for the contract to be valid. The most important conditions are:
  - *Distribution specification*: kind of distribution to be performed, as described in Section 5.3.3.

  - *Validity specification*: validity of the contract and thus of the code distribution, which is normally expressed as a time interval.

- *Signature*: the contract is signed by the contract issuer.

This contract is expressed in XACML [xac05]. XACML is an OASIS standard, which provides an XML-based language to express generic access control (or authorisation) policies, and standard messages for a query/response protocol. More precisely, the contract is expressed as an XACML policy so it can be directly used in the policy enforcement process. Figure 5.5 shows an example code owner contract policy, where an ACDS administrative authority establishes a contract with a given code owner, both identified by a hash of their public key, with the following information: the code owner (subject) can distribute code with $CGID = 1f332e209d0dc354$ during one year; since no action is specified the code owner can perform all the possible actions.

Summarising, the policy specifies a target which is the code owner and provides a set of rules. Each rule can be applied to an specific target, which narrows the policy's target, normally by specifying a resource or an action and describing some conditions. It is important to note that in the previous example of Figure 5.5, we are using the XACML 3.0 administration specification to include the issuer in the policy, which is currently not approved as a standard. The current XACML standard (version 2) does not support the policy issuer element although it may be included apart of the XML digital signature provided with the contract.

Contract Enforcement Module

The enforcement of the code owner contract as well as other security related policies are provided by CEM in each PoP (see Figure 5.2). This module is mainly responsible for the following tasks:

- Authentication of the code owner and code provider operations (upload, deletion, modification, ...), and the code owner contract.

- Enforcement of the code owner contract, PoP local policies, and other generic policies (provided globally or on a region-base by ACDS).

```
<Policy PolicyId="ACDS-contrat-policy:example"
        RuleCombiningAlgId="deny-overrides">
  <PolicyIssuer>
    <Attribute
        AttributeId ="subject:subject-id"
        DataType="xmldsig#RSAKeyValue">
      <AttributeValue>
        <RSAKeyValue>
          <!-- ... -->
        </RSAKeyValue>
      </AttributeValue>
    </Attribute>
  </PolicyIssuer>
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="rsakeyval-equal">
          <AttributeValue
              DataType="xmldsig#RSAKeyValue">
            <RSAKeyValue>
              <!-- ... -->
            </RSAKeyValue>
          </AttributeValue>
          <SubjectAttributeDesignator
              AttributeId = "subject:key-info"
              DataType="xmldsig#RSAKeyValue"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources><AnyResource /></Resources>
    <Actions><AnyAction/></Actions>
  </Target>
  <Rule RuleId="validity period rule" Effect="Deny">
    <Target>
      <Subjects><AnySubject /></Subjects>
      <Actions><AnyAction /></Actions>
      <Resources>
        <Resource>
          <ResourceMatch
              MatchId="anyURI-equal">
            <AttributeValue
                DataType="XMLSchema#anyURI">
              urn:acds:cgid:1f332e209d0dc354
            </AttributeValue>
            <ResourceAttributeDesignator
                DataType="XMLSchema#anyURI"
                AttributeId ="resource:resource-id"/>
          </ResourceMatch>
        </Resource>
      </Resources>
    </Target>
    <Condition FunctionId="and">
      <Apply FunctionId="date-greater-than-or-
          equal">
        <Apply FunctionId="date-one-and-only">
          <EnvironmentAttributeSelector
              DataType="XMLSchema#date"
              AttributeId ="environment:current-date"/>
        </Apply>
        <AttributeValue DataType="XMLSchema#date">
          2008-01-01
        </AttributeValue>
      </Apply>
      <Apply FunctionId="date-less-than-or-equal">
        <Apply FunctionId="date-one-and-only">
          <EnvironmentAttributeSelector
              DataType="XMLSchema#date"
              AttributeId ="environment:current-date"/>
        </Apply>
        <AttributeValue DataType="XMLSchema#date">
          2009-01-01
        </AttributeValue>
      </Apply>
    </Condition>
  </Rule>
</Policy>
```

Figure 5.5: Example of code owner contract policy.

These tasks are performed when a code owner requests an operation on a PoP of the network, such as *upload*, *update*, or *delete* a given code. Figure 5.6 shows the main components of CEM and their main tasks.

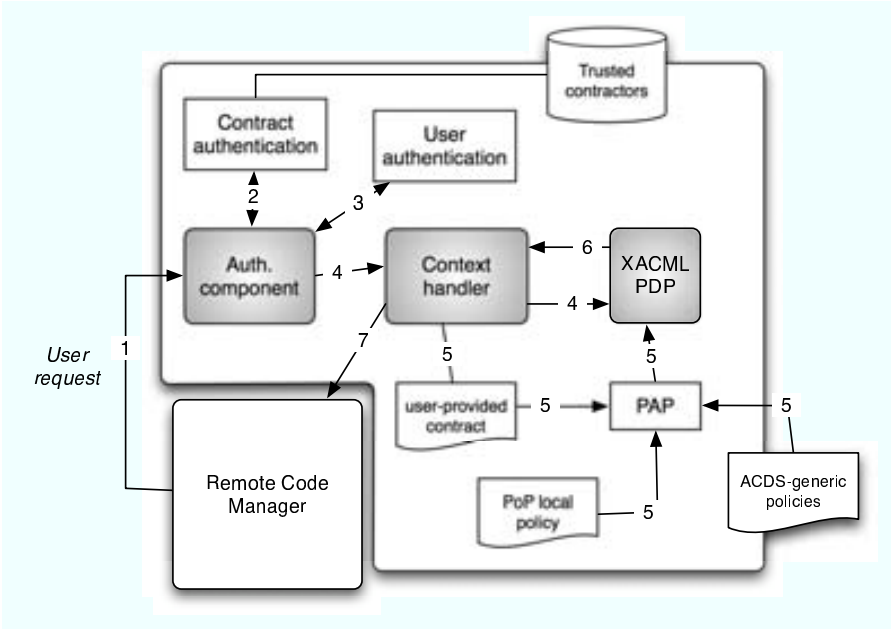

Figure 5.6: Contract Enforcement Module overview.

To see how this module works and its functionality we show the enforcement process of a generic code owner CO requesting to $upload$ a given code $CGID_0$. The process will normally consist of:

1. The module receives the code owner request, which may include the code owner contract if it is the initial code upload.

2. If needed, the Authentication Component (PoP-AC) authenticates the code distribution contract by verifying its signature with a repository of trusted ACDS administrative authorities keys. These keys are distributed to all nodes (normally making use of certificates in a PKI-like fashion). The contract provides the code owner public key, which will be used to authenticate the operation request on the specific code.

3. PoP-AC authenticates the code owner operation request with the hash of the public key present in the contract as the subject.

4. The request is passed to the Context Handler component (PoP-CH). This component formats the request as a valid XACML request message and passes the contract to the Policy Administration Point (PoP-PAP) to be used in the XACML based Policy Decision Point (PDP).

5. PoP-PAP combines the following XACML policies:

   - Code distribution contract as provided by PoP-CH.
   - PoP-local policy: each PoP may impose constraints locally on its resources, either temporarily or permanently.
   - ACDS-generic policy: PAP includes references to ACDS-generic policies maintained by the ACDS administration authorities. These policies may have a global or region-based applicability.

6. XACML PDP is the component that actually performs the authorisation decision. It is a generic XACML PDP, since both, the user request and the policies are standard XACML request and policies.

   PDP outputs a response to PoP-CH. This response can be: *permit* (if access is allowed), *deny* (if access is denied), *indeterminate* (if an error occurred or some required value is missing), or *not applicable* (if no policy applies to the request).

7. PoP-CH reports the result to the main PoP-RCM.

5.3.5 Service Interface

The ACDS Interface allows code users and code owners to use the service without knowledge of its internal details. The following sections state the requirements of the interface and propose a specification based on the REST model [FT02].

Requirements

A set of operations must be supported by the interface in each PoP node. These operations, which are used by code users and code owners, are bound to the CGID and Code Identifier (CID) elements. The CGID element must accept the following operations:

- Create a CGID element with or without a specific code distribution contract.
- Get the associated code distribution contract.
- Update the associated code distribution contract.
- Get a list of available codes (CIDs) in the node.
- Remove the CGID element.

Some of these operations refer to the code distribution contract. In case no contract is given in the CGID creation, no code owner can be assigned to the code and a specific anonymous contract is automatically created by the service. In this case only the on-demand type of service can be offered. Regarding the CID, the operations accepted are the following ones:

- Upload agent code and its signature.
- Update existing agent code and its signature.
- Download agent code and its signature.
- Remove agent code.
- Get an agent code profile and its signature.

Operations that involve the publication, modification or removal of agent codes and their properties require authentication. Nevertheless, the non authenticated publication of agent codes is possible using the anonymous service functionality.

Since ACDS is a global service focused on an efficient distribution of agent code, there is a set of basic requirements that must be fulfilled, specially on the interface provided to the clients. Therefore the requirements for the ACDS Interface are: simplicity,

scalability, low latency, efficient data transference, portable interface, and support for any type of code.

REST Interface

The ACDS Interface can be implemented using different technologies such as SOAP, XML-RPC, and REST. Actually, several complementary interfaces can be provided. Nevertheless, in this chapter we have focused on only one interface based on the REST technology [FT02]. REST is a coordinated set of architectural constraints based on the HTTP protocol. It is characterised for minimising the communication latencies, simplifying the network communication, and maximising the independence and scalability of component implementations.

Interfaces implemented with REST have a base address, a set of elements coming from this base address (in our case the CGID and CID), and several operations associated to each element. The REST interface which is proposed is detailed in Table 5.1.

Base URL:	*http://server/acds/{cgid}/{cid}*					
Element	**Method**	**Req. Param.**	**Req. Headers**	**Resp. Headers**	**Body**	**Functionality**
cgid	PUT		[Authorization], [Depth]		[XACML contract]	Creates a code group with or without an assigned contract. The Depth header indicates the depth in the neighbourhood tree (it is used to control when the code propagation reaches the established neighbourhood degree).
	DELETE		Authorization			Removes a code group and its content, i.e., its codes.
	GET	list, contract			List of codes or XACML contract	Get information from the code group: list of its content and associated contract.
	POST		Authorization	XACML contract		Update the contract associated to the code group.
cid	PUT		[Authorization], SR, HCID, Code profile, [Code signature], [Code profile signature]		Code	1.Upload a specific code (CID) with an associated profile. 2.Update a specific code. Note: A specific Code signature and Code profile signature headers have been included to deal with code and profile signatures. These headers are optional and are encoded in Base 64.
	DELETE		Authorization			Removes a specific code (CID).
	GET			Code profile, [Code signature], [Code profile signature]	Code	Download a specific code (CID) and profile.
	HEAD			Code profile, [Code profile signature]		Get a specific code profile.

Table 5.1: REST Interface.

Operations that require authentication take advantage of the "Authorization" header defined in the RFC 2617 [BLFMb]. The authentication method proposed for the described interface is based on the signature of the MD5 hash [Riv] of a string that contains several parameters of the HTTP request:

$$\begin{aligned} \textit{String To Sign} \;=\; & \textit{HTTP Method} + \text{“}\backslash n'' + \textit{HTTP Request URI} + \text{“}\backslash n'' + \\ & + \textit{Date} + \text{“}\backslash n'' + \textit{Content MD5} + \text{“}\backslash n'' + \\ & + \textit{Content-Type} + \text{“}\backslash n'' + \textit{Interface HTTP Headers} \end{aligned}$$

Notice that the HTTP headers defined in the interface are also included sorted in the appearance order. The string is encoded in UTF-8 and the result of its hashing is encoded in Base 64. The signature is put in the "Authorization" header together with the subject of the certificate, which is the identification of the certificate owner (in this case the hash of its public key). The format of this header is:

Authorization: PublicKeyAuth *certificate-subject* : *signature*

where the "PublicKeyAuth" states the authentication type, the *certificate-subject* identifies the subject of the operation, which may be the code owner or the code provider, and the *signature* is the chain encoded in Base64 that contains a PKCS-7 [Lab93] with the signature of the previously described *String to Sign*. This methodology is similar to the one used in the Amazon webservices [AWS].

5.4 Implementation and performance

5.4.1 Service implementation

A preliminary version of ACDS has been implemented [acd] to evaluate the service proposed in this chapter. It has been developed in Java 6 taking advantage of the Java integrated HTTP server and the Java API for RESTful Web Services (JAX-RS) [HS07], which is implemented by the Jersey Java project.

Furthermore, a Java ACDS client library has also been developed to easily integrate the service into existing AMs. The library has been used in the Push Cache Transfer Protocol (PCTP) and REST Transfer Protocol (RESTTP) protocols of the Inter-Platform Mobility Architecture (IPMA) to perform the tests explained in the next section.

5.4.2 Performance tests

A set of performance tests have been done to evaluate the advantages of ACDS using the JADE Inter-Platform Mobility Service (JIPMS). Since the messaging system implemented in the JADE AM is not devised to send much data in a single Agent Communication Language (ACL) message [Cuc04, JJK06], the tests have not been carried out only with PCTP, but also with RESTTP which does not use large ACL messages. The evaluation setup and test suite used to run the tests are the same of Section 4.2. Three different scenarios, similar to the ones shown in Sections 4.4 and 4.5, have been used to compare the migration performance with and without the ACDS enabled. The network conditions of each scenario have been also simulated by using the NetEm [Hem05] Linux utility, which allows the adjustment of network latency, bandwidth, and packet loss.

A set of agents with code sizes of 5KB, 10KB, 25KB, 50KB, 100KB, 250KB, 500KB, and 1000KB packed in uncompressed JAR files have been used to perform the tests. Each test has been done with one and, later, ten migrating agents of a specific size in each location. As in Chapter 4 the measures shown in the tables are the average time consumed by the agents from the AM. The itinerary of the agents only involves two locations and it is repeated a fixed number of times (iterations or migration round-trips). In these tests agents smaller or equal than 100KB repeat their itinerary 100 times, while larger agents, which spend more time per migration, repeat their itinerary only 10 times. The local cache mobility mechanisms have been completely disabled, therefore although agents only visit two real locations their behaviour can be extrapolated to the case of agents that visit a new location in each migration. Agent codes used in the tests are delivered to each PoP in advance.

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
PCTP	1	126	140	222	493	1,734	11,943	41,499	150,162
ACDS Off	10	77	85	140	339	1,200	8,300	32,710	116,721
PCTP	1	100	101	103	113	123	161	211	320
ACDS On	10	63	63	68	70	77	106	154	253

Table 5.2: Scenario 1: Agents moving in the same region (PCTP) (in ms).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	101	102	106	109	126	160	212	311
ACDS Off	10	71	71	73	77	87	108	145	242
RESTTP	1	104	106	107	112	131	165	217	304
ACDS On	10	69	69	70	73	84	106	152	245

Table 5.3: Scenario 1: Agents moving in the same region (REST) (in ms).

Scenario 1: Agents moving in the same region

The first scenario (see Figure 5.7) is composed of only one ACDS region with two APs, i.e., there is only one PoP node providing service to the APs. The response time considered between all the hosts of the region is less than 1ms, there is no loss of packets, and there is a bandwidth of 100 Mb/s. This is a typical scenario of a Local Area Network (LAN).

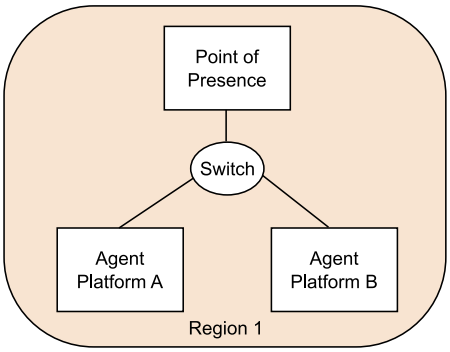

Figure 5.7: Scenario 1.

Tables 5.2 and 5.3 show the performance results obtained in this first scenario. And Figures 5.8 and 5.9 show a graphical representation of them.

In the case of PCTP the results are clearly favourable to the tests with the ACDS enabled. In this specific scenario, where the access conditions to any host of the network are the same (included the PoP), the ACDS improvement is consequence of the penalty

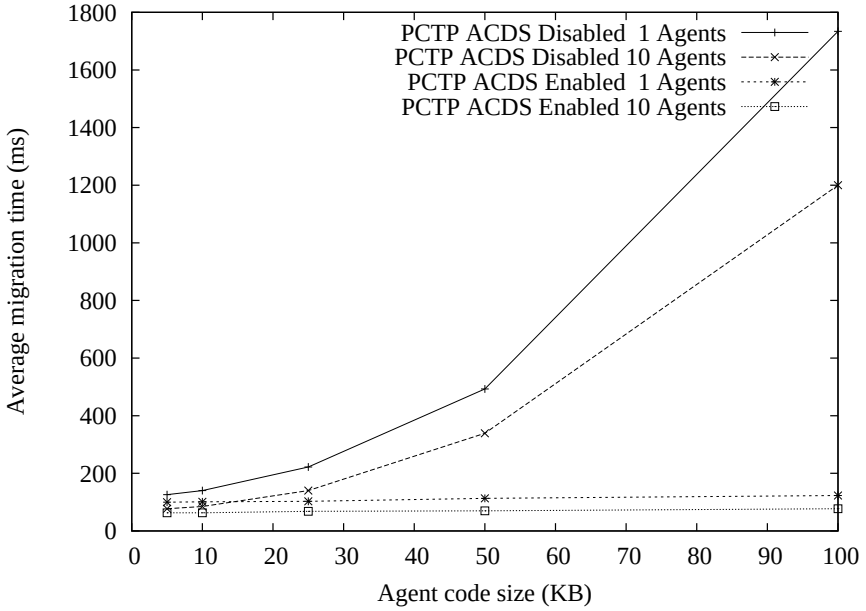

Figure 5.8: Performance Scenario 1 (PCTP).

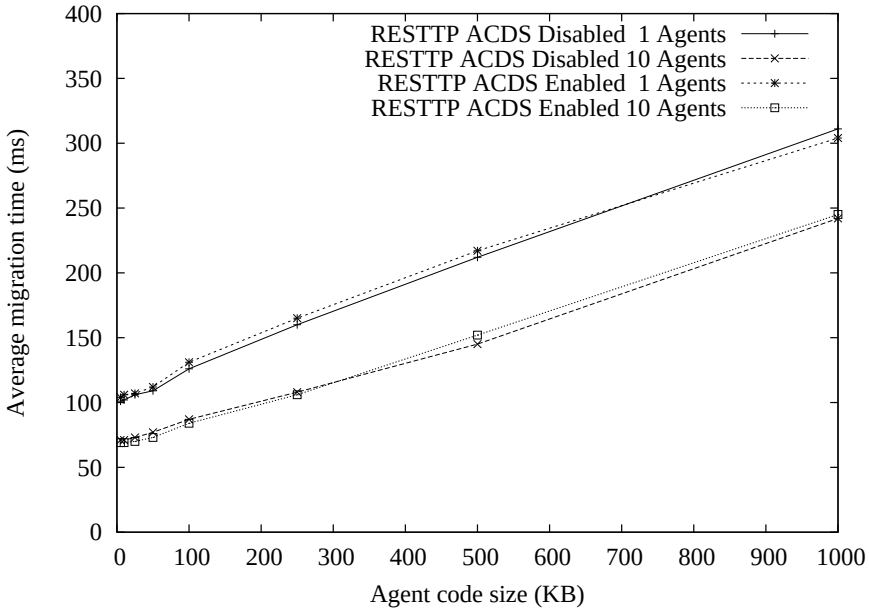

Figure 5.9: Performance Scenario 1 (RESTTP).

imposed by the ACL data transmission implemented in JADE (see [Cuc04, JJK06]), where the time spent in these agent migrations increases exponentially. Notice that only agents with codes up to 100KB are shown, since the large average migration time differences complicate the comparison of the results.

In the case of RESTTP, which as it is explained in Section 4.5 is more efficient than PCTP, the results with and without the ACDS enabled are almost the same, since the network conditions and the technologies used are also the same.

Therefore the REST model, used in RESTTP and in ACDS, has resulted highly efficient in data transmission (as it was required in Section 5.3.5). In the best case, using RESTTP with the ACDS enabled, there is only a difference of approximately 176ms between the smallest and the biggest agent (codes between 5KB and 1000KB), which means a transmission rate of approximately 11MB/sec (close to the theoretical maximum data transmission rate of the network devices used). With the ACDS enabled the time spent in the migration increases linearly. Taking into account the results presented, in concurrent migrations with 10 agents the average time spent per agent decreases around a 70% of the time spent with a single agent migration.

Scenario 2: Agents moving between different regions

The second scenario (see Figure 5.10) is composed of two ACDS regions with an AP in each one, i.e., there are two PoP nodes providing service to an AP in each region. The response time considered between all hosts in a region is less than 1ms, there is no loss of packets and there is a bandwidth of 100 Mb/s, i.e., these are the conditions between the AP and its correspondingly PoP. Regarding the communication between the two regions, two cases have been simulated. The first one supposes an error free transmission with no packet loss. In this case, the response time considered between the two regions is 120ms with a variation of 10ms and a bandwidth of 100 Mb/s. The second case supposes the same parameters with a 5% loss of the packets transmitted. The scenario and the two cases described are representative of real Wide Area Networks (WAN).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
PCTP	1	2,017	2,264	2,719	3,423	5,216	16,496	46,232	153,689
ACDS Off	10	1,544	1,695	2,094	2,303	2,550	9,133	33,870	112,927
PCTP	1	1,604	1,587	1,629	1,663	1,705	1,836	2,076	2,483
ACDS On	10	1,086	1,083	1,086	1,090	1,095	1,102	1,139	1,248

Table 5.4: Scenario 2: Agents moving between different regions (no packet loss) (PCTP) (in ms).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	2,502	2,727	3,027	3,451	3,998	5,037	6,128	7,868
ACDS Off	10	1,315	1,365	1,527	1,758	3,590	4,743	5,822	7,558
RESTTP	1	1,741	1,742	1,750	1,749	1,808	1,831	1,854	1,944
ACDS On	10	1,023	1,024	1,022	1,023	1,039	1,191	1,154	1,164

Table 5.5: Scenario 2: Agents moving between different regions (no packet loss) (RESTTP) (in ms).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
PCTP	1	2,988	3,248	4,276	5,932	9,673	27,001	67,196	197,131
ACDS Off	10	2,189	2,424	3,289	4,284	5,123	16,034	43,207	135,615
PCTP	1	2,083	2,089	2,038	2,105	2,139	2,275	2,124	2,292
ACDS On	10	1,479	1,489	1,485	1,484	1,495	1,497	1,488	1,476

Table 5.6: Scenario 2: Agents moving between different regions (5% packet loss)(PCTP) (in ms).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP	1	4,382	4,656	5,033	5,505	7,160	10,760	20,019	38,209
ACDS Off	10	2,707	2,746	2,844	3,269	4,275	11,114	20,548	37,354
RESTTP	1	2,708	2,754	2,754	2,624	2,722	3,463	3,053	3,907
ACDS On	10	1,640	1,616	1,636	1,599	1,615	1,825	1,783	1,703

Table 5.7: Scenario 2: Agents moving between different regions (5% packet loss) (RESTTP) (in ms).

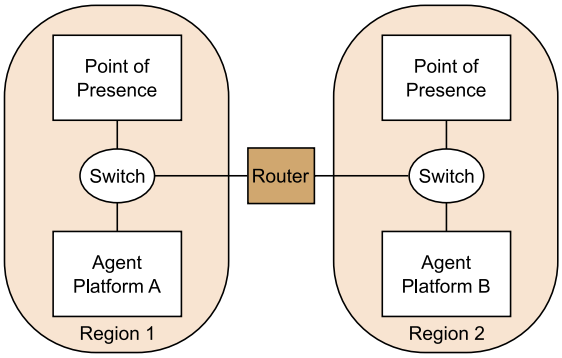

Figure 5.10: Scenario 2.

Tables 5.4 and 5.6 for PCTP, and Tables 5.5 and 5.7 for RESTTP, show the performance results obtained in the two network environments of this second scenario. Figures 5.11 and 5.13 for PCTP, and Figures 5.11 and 5.13 for RESTTP show a graphical representation of them.

In the first case, the WAN without network loss (see Table 5.4), the increase of the response time considerably changes the performance of agent migrations. An agent migration may be up to 20 times longer than in the first scenario. Nevertheless, there is an exception with migrations of agents with large codes using PCTP with the ACDS disabled, in this case the time spent in each migration is similar because of the ACL message penalisation (see Section 4.4.2). When the ACDS is enabled, since the code request is not subject to the WAN access time, the time increase is only associated to the ACL message handshake, e.g., see that in case of 10 migrating agents with PCTP, in a similar way than the first scenario, there is only a difference of approximately 160ms between the smallest and the biggest agent, i.e., the time spent in the code transmission has not increased.

In the second case, the WAN with a loss of 5% of the packets (see Table 5.6), the performance of the agent migrations decrease regarding the tests presented up to now. Compared with the previous case, the average time spent in each agent migration with the ACDS disabled increases from a minimum of 20% to a maximum of 100% more with PCTP, and from a minimum of 20% to a maximum of 400% more with RESTTP. When the ACDS is enabled the time spent does not increase more than a 40% with PCTP, and a 100% with RESTTP. Therefore, the use of ACDS is rather appropriate

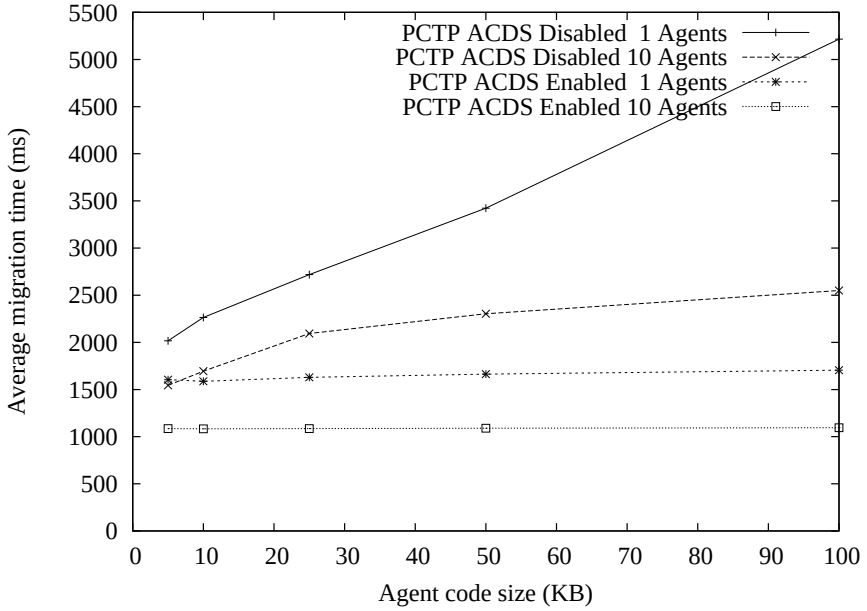

Figure 5.11: Performance Scenario 2 (no packet loss) (PCTP).

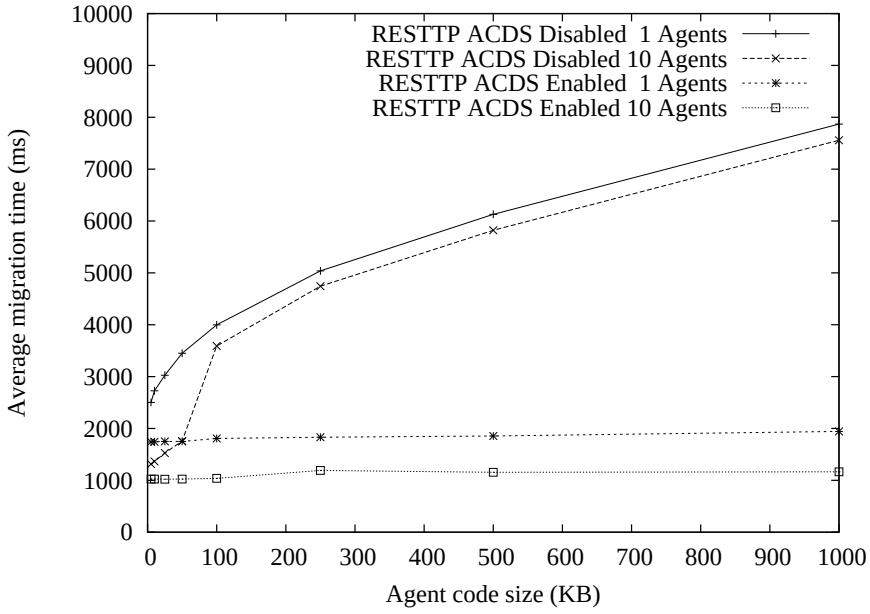

Figure 5.12: Performance Scenario 2 (no packet loss) (RESTTP).

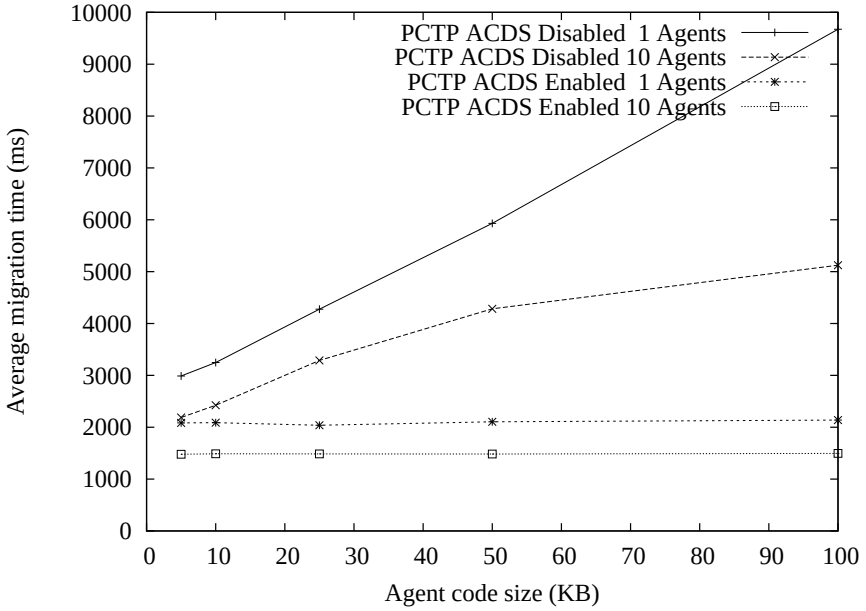

Figure 5.13: Performance Scenario 2 (5% packet loss) (PCTP).

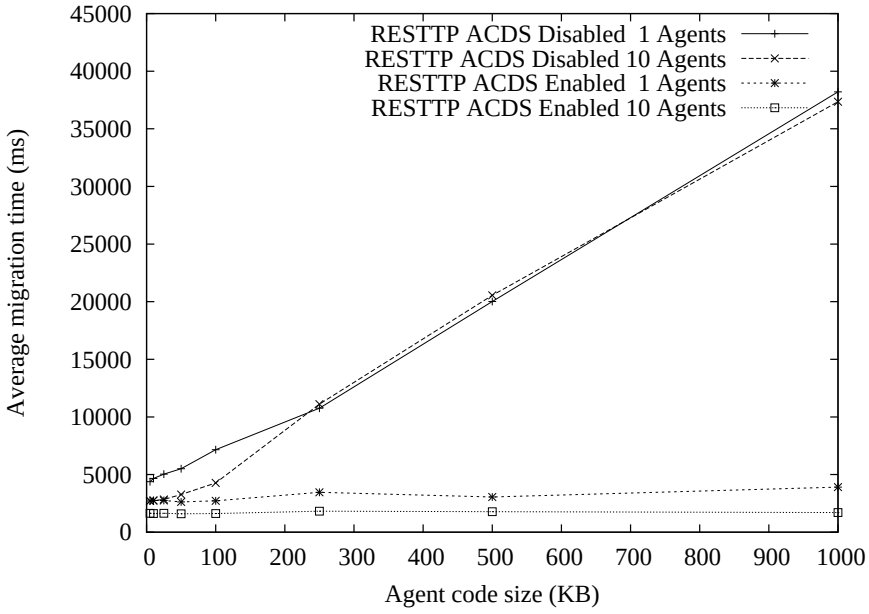

Figure 5.14: Performance Scenario 2 (5% packet loss) (RESTTP).

when the network conditions get worse.

The penalisation imposed by the response time increase can be mitigated by the use of concurrent migrations. Usually, these ones get a better throughput than single agent migrations. When the ACDS is disabled, concurrent migrations consume approximately a 70% of the time than single migrations. When the ACDS is enabled, concurrent migrations consume approximately a 65% of the time with PCTP, and a 60% of the time with RESTTP. These values slightly change depending on the agent size.

It is worth comparing the results obtained with PCTP and RESTTP protocols with the ACDS enabled. In the first case, no packet loss, there is no big differences, for agents with codes of 5KB to 100KB PCTP tends to perform better with single agent migrations, and RESTTP with concurrent ones. For larger agents this tendency is inverted. But, in the second case, 5% packet loss, PCTP performs better in all the cases. This is because RESTTP establishes an additional connection to retrieve the agent data, which in PCTP is included in the first ACL message sent. Nevertheless, in case the data size was larger, this tendency would be inverted.

Scenario 3: Asymmetric links

The third scenario (see Figure 5.15) is composed of one ACDS region with two APs, i.e., there is one PoP node providing service to two APs. All the APs in this scenario use a 20Mb/s (downstream) 1Mb/s (upstream) ADSL link to the Digital Subscriber Line Access Multiplexer (DSLAM). The response time associated to these links is around 30ms. Data exchanged between APs, since traverses two ADSL links, can only reach 1Mb/s of bandwidth and the response time is 60ms. On the other hand, data received from the PoP node in the APs can reach 20Mb/s of bandwidth with a response time of 30ms. Obviously, uploaded data can only reach 1Mb/s, although there is less data to upload than to download.

Table 5.8 for PCTP and Table 5.9 for RESTTP show the performance results obtained in the third scenario. And Figures 5.16 and 5.17, for PCTP and RESTTP respectively, show the graphical representation of them. The results are also favourable to the tests which have been performed with the ACDS enabled. This is because of the higher download rate from the ACDS PoP (20Mb/s). Nevertheless, because of the new

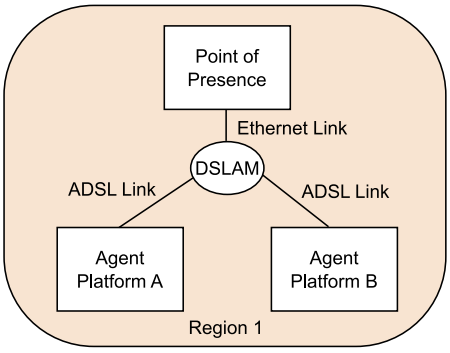

Figure 5.15: Scenario 3.

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
PCTP ACDS Off	1	1,008	1,100	1,675	2,496	5,002	19,412	56,628	177,940
	10	801	882	1,253	1,493	2,714	12,205	39,581	125,591
PCTP ACDS On	1	972	952	1,083	1,191	1,298	1,468	1,708	2,140
	10	538	539	542	547	603	769	998	1,409

Table 5.8: Scenario 3: Asymmetric links (PCTP) (in ms).

Protocol	N.Agents	5KB	10KB	25KB	50KB	100KB	250KB	500KB	1000KB
RESTTP ACDS Off	1	1,357	1,443	1,683	2,131	2,972	5,670	10,134	18,998
	10	701	742	1,012	1,224	2,271	5,532	9,995	18,868
RESTTP ACDS On	1	1,183	1,138	1,213	1,285	1,424	1,604	1,825	2,268
	10	540	542	566	608	724	905	1,121	1,471

Table 5.9: Scenario 3: Asymmetric links (RESTTP) (in ms).

link conditions, there are migrations which are performed 10 times slower than in the first scenario. In this case, with the ACDS enabled, the difference between the smallest and the biggest agent migration round-trip (in case of 10 concurrent agent migrations) are 871ms for PCTP and 931ms for RESTTP. These results are approximately 5 times higher than the ones obtained from the scenario 1, i.e., in this case the practical bandwidth is around 2 MB/s.

The most affected part by the new link conditions are the message handshakes. See the exchange of ACL messages to perform the agent migration of a single agent with PCTP. In this case, the time spent is around 1000ms for each migration round-trip. This time is coherent with the new link conditions, which has a response time of 60ms. In the first case of the scenario 2 the response time associated to the link was 120ms and, therefore, the time spent exchanging ACL messages was about 2000ms. This is similar

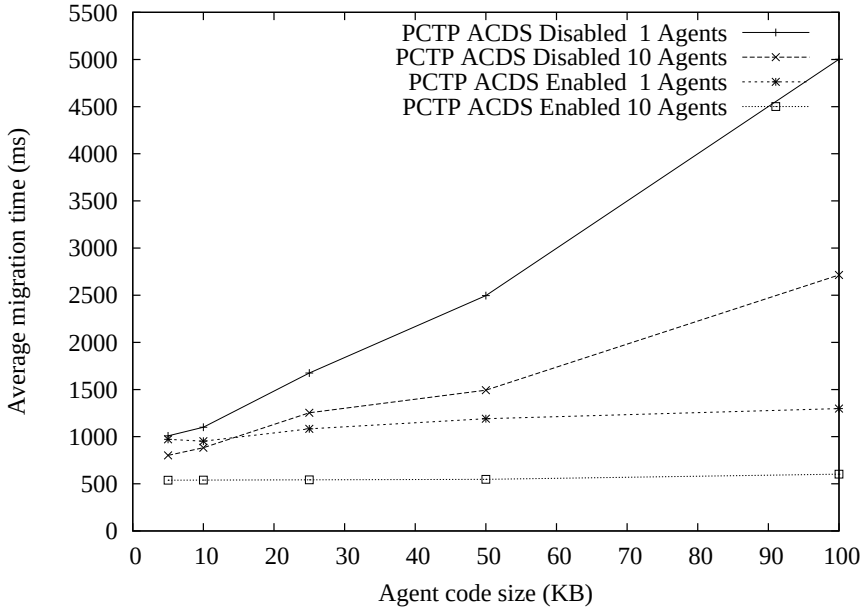

Figure 5.16: Performance Scenario 3 (PCTP).

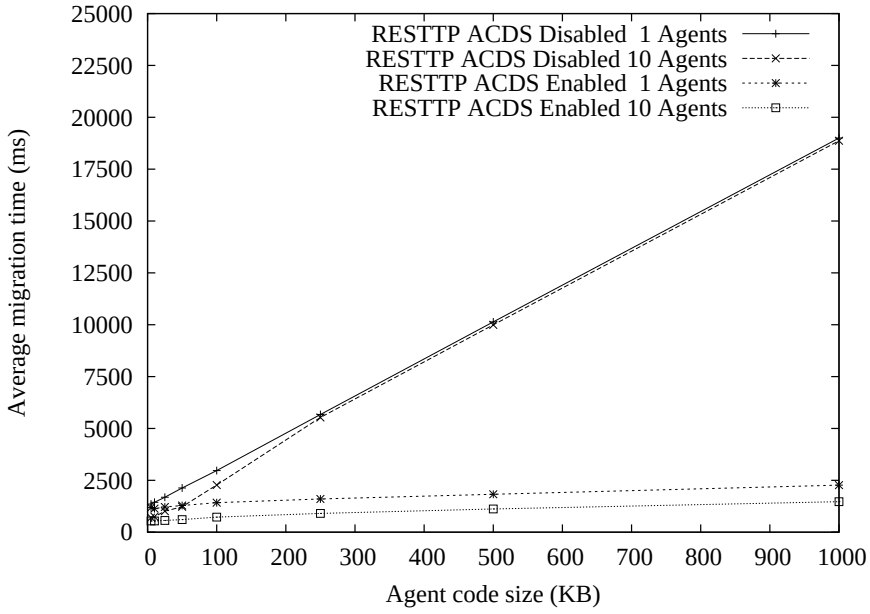

Figure 5.17: Performance Scenario 3 (RESTTP).

with RESTTP.

In all the tests, the use of parallelism with 10 agents has meant migration times around the 70-75% of the single agent migration time. But because of the fact that the link with the ACDS PoP is also subject to a higher response time than the other scenarios, the use of parallelism even obtains a greater improvement (around a 50% of the single migration time). This is because the transmission of data can be done in parallel with the ACL message interaction of the other migrations. In other scenarios, although this can also be done, the data transmission time is shorter and the effect of the migration pipeline is not so noticeable.

Regarding PCTP and RESTTP with the ACDS enabled, PCTP performs better in all the cases. Similar to the previous scenario, this is because RESTTP establishes an additional connection to retrieve the agent data, which in PCTP is included in the first ACL message sent.

Conclusions

The tests performed denote that in all the cases the best results have been obtained when the ACDS service was enabled. Furthermore, in case of unfavourable network conditions the use of the service is also highly encouraged, since the benefits of the code caching service are even more noticeable.

Another remarkable fact is that the use of the ACDS when concurrent agent migrations take place allows an even better average migration time. This is because the different mechanisms involved in the agent migration, and the mechanisms to request the agent code to the PoP are arranged as a pipeline and can work in parallel, since they are completely independent among each others.

Finally, regarding the use of one or another migration protocol (PCTP or RESTTP in this case) in combination with the ACDS, in the tests presented it seems that PCTP performs better. But this exclusively depends on the way the protocols deal with the agent data and state.

5.5 Related work

In this chapter a code distribution service, initially conceived for Mobile Agents (MAs), has been proposed. No other initiatives in the literature have been found with this exact purpose and functionality, but there are some works that cover specific areas of the service proposed. This is the reason why the related work is presented in several categories.

5.5.1 Code distribution

The distribution of agent codes has not been already proposed as an independent service such the one presented in this chapter. Nevertheless, as each AM has its own agent code distribution mechanism, several works exist in the area. Nicklisch *et al.* [NQKA98] states three types of code distribution: *migration*, *push*, and *pull*. According to their definition, the *migration* distribution is the one which sends the agent code along the agent state during an agent migration. This is the most common way to distribute the code and it is supported by most of the AM, such as Aglets [LM98], SeMoA [RJS01], Tracy with Kalong [BR05], and JADE with IPMA (see Chapter 3). The *push* code distribution consists of sending the agent code to a set of locations, sometimes associated to the agent itinerary, before the agent launch. This scheme has been used by Nicklisch *et al.* [NQKA98] in INCA and, with several improvements in the distribution list, by Gavalas *et al.* [GGGO02] in their hierarchical network management solution based on MAs. And the *pull* code distribution consists of downloading the code once the agent reaches a location. This scheme, usually in combination with the migration scheme, is rather extended. It is supported by the same AMs which have been previously listed with the push distribution.

The service proposed in this chapter supports the push and pull schemes (see Section 5.5.1). The difference with all the initiatives mentioned is that ACDS is a standalone service that is not bound to any specific AM. Furthermore, it can manage several codes per agent, which can be developed in any PL, it can deal with code profiles and code signatures, it supports access control regarding the operations to manage the code, and it provides a standard interface for its usage.

5.5.2 Content Delivery Networks

A Content Delivery Network (CDN) [PB07] is a network composed of specific nodes, called PoPs in this thesis, close to Internet backbones that ease the content distribution. CDNs are focused on the efficient distribution and caching of content. Usually, there is an important hardware infrastructure behind. The performance experienced by users that access content present in a CDN is improved, while the saturation of wide area networks is reduced, see [RKSB05] for more details. In many cases the service is transparent to its users, i.e., they do not notice they are using it.

There are several CDNs dedicated to different types of content. The most well-known CDN is Akamai [DMP+02, SLBW05] for web content. Although web content can include code, such as applets, no special contract and versioning facilities are supported. There is a CDN specifically designed to distribute applications, the AppStream [App]. Nevertheless it is based on a centralised approach conceived for small environments. There are other initiatives similar to this, such as the ASP-NG system [BGN+05] which serves applications on demand, although it is restricted only to J2EE applications.

5.5.3 Peer to Peer networks

Peer to Peers (P2Ps) are distributed networks that maintain information, usually files, through several nodes. A decentralised model is generally used. Each node is considered an equal, and has the same properties and functionalities as others. P2Ps are focused on file distribution, are fault tolerant in most of the cases, and can prevent server bottlenecks. Examples of P2Ps are the well-known Gnutella [RFI02], FastTrack [LRW03], JXTA [Gon01], and eDonkey [Tut04]. Unfortunately, P2Ps are not focused on offering a high performance data transmission, they do not provide versioning support, and they increase the general bandwidth consumption.

5.5.4 Distributed Version Control systems

Distributed Version Control systems are applications that manage and keep source code, which is distributed through several hosts, in different revisions and branches. Therefore, the information is usually structured in a hierarchical representation. The benefits of these systems are that the code can be distributed and replicated through several nodes, and that there is a clear file version maintenance policy.

Examples of them are GIT [git], Darcs [Rou05], and Mercurial [O'S]. Regarding our purposes, these systems are too focused on source code distribution instead of their binaries, have poor authentication methods, which are usually based on secure shells such as the SSH, and are devised to maintain files which are continuously changing.

5.6 Conclusions

The advantage, and at the same time inconvenient, of MAs is the fact of carrying the agent code, data, and state everywhere they go. The degree of autonomy agents have as a result of this characteristic is exceptional regarding other technologies. But carrying these components may be an inconvenient when they weight considerably.

In this chapter ACDS, which aims to reduce the penalisation imposed by the mentioned agent structure, has been presented. ACDS distributes the agent codes to a subset of regions in which the network is divided. The agent codes are distributed under contract, i.e., agreeing several specific conditions and possibly having to pay for the service, or anonymously without having to pay for the service. In the first case the agent code is deployed in advance to a set of regions enforced by contract. In the second case, since no regions to deploy the agent code are known in advance, the code is spread on-demand to each PoP. Therefore, the service benefits are not always guaranteed in this last case.

Contracts are expressed in XACML and there is an access control mechanism which enforces a set of agreed restrictions to the basic ACDS operations. Therefore, it is guaranteed that a code owned by a specific person will be distributed according to the parameters agreed, and that no one will be able to modify this code or its distribution. The code management is very flexible and it is self-administrable by each code owner.

Regarding the interaction with the service a REST interface has been proposed. Although different technologies can be used to accomplish this purpose, REST has been chosen since: it is independent of the PLs used by the service and client implementations, it is standard, it is efficient, and it has been extensively tested in other applications.

Finally, the benefits of the service proposed have been demonstrated. A set of performance tests comprising different scenarios which are present in the real world have been done (a local area network, a wide area network with and without packet loss, and a network composed of several ADSL lines).

Chapter 6

Interoperability

For mobile agents to be deployed on Internet scale distributed systems, interoperability with different types of agent middleware needs to be ensured. This chapter presents several proposals to achieve the challenge of agents freely migrating in heterogeneous environments.

6.1 Introduction

According to Pinsdorf *et al.* [PR02], two Mobile Agent Systems (MASs) are interoperable if a Mobile Agent (MA) can interact and communicate with other agents (local or remote), and if the agents of one system can migrate to the other system, i.e., they can leave their system and resume their execution in the next interoperable system. This kind of interoperability is called *full interoperability* and can be tackled from the four areas described in Section 2.4.1.

Despite the fact that there are several specific proposals for each area, a full interoperability mechanism general enough to cover all the possible combinations of Agent Middlewares (AMs), Programming Languages (PLs), and Underlying Architectures (UAs) is hardly achievable. Too many variables are involved. Nevertheless, full interoperability can be reached assuming some limitations with the intended scope. The most common limitation in the literature is the entailment to a specific PL and UA, usually Java and its Virtual Machine (VM) [FGR07]. Another issue with the existing

mechanisms is that they do not usually support currently in-use agent standards, such as MASIF [OMG97] or the IEEE-FIPA [FIP02a] specifications.

In order to face the shortcomings of the existing approaches, three proposals are presented in the next sections. These solutions are specifically focused on the *middleware*, and *programming language and underlying architecture* areas. A full mobile agent interoperability within the IEEE-FIPA context is achieved by the combination of these proposals together with the Inter-Platform Mobility Architecture (IPMA) described in Chapter 3, that covers the *mobility* area, and the IEEE-FIPA specifications, that cover the *communication* and part of the *middleware* areas.

- **Common Agent Interface (CAI):** A common abstract agent interface is proposed in Section 6.3 to deal with the interoperability in the middleware area. The idea behind it is to set the basis for a minimal interoperable agent compliant with the IEEE-FIPA specifications and suitable for any PL. Agents can be moved to any AM supporting CAI and the specific PL in which they are developed (this was firstly considered in [CMR+07, COO+07]). Therefore, the middleware area of interoperability is dealt in this section.

- **Multiple Execution Environments:** Assuming a common agent interface such as CAI, the idea of having multiple Execution Environments (EEs) for a selection of PLs within an AM is discussed in Section 6.4. This allows the execution and local interaction of agents present in the same AM, but developed in different PLs. Therefore, the programming language and underlying architecture area of interoperability is dealt in this section. This is an idea similar to the one applied in [JLvR+02, GCK+02, Pei02, CCP06, OB04].

- **Multiple Code Agents:** Agents with several equivalent versions of their code, each one written in a different PL, are proposed in Section 6.5. These agents can visit any Agent Platform (AP) with support for one of the PLs of their code. They keep and resume their state although using one different code in each visited location. A shared data representation mechanism is proposed. Several data representation languages are discussed to agree on a common PL independent way

of representing the agent data. This approach is complementary to the previous one and deals with the same area of interoperability.

The adoption of one, two or three of the proposals implies different grades of constraints and complexity. Therefore, MAS developers must choose the most appropriate ones to their needs. It is worth mentioning that these proposals are only devised to support weak agent mobility. The use of strong mobility impose serious restrictions to the agent interoperability because of its strong dependency with the UA [CLZ00]. See Section 2.3.2 for more information about types of agent mobility.

Finally, the use of the these approaches in combination of IPMA and the IEEE-FIPA specifications ends up with a full interoperable solution for MASs in the IEEE-FIPA context. In Section 6.6 several proposals present in the literature claiming to provide agent interoperability are discussed.

6.2 Background

Proposing a full interoperability solution for MASs is a task that comprises the study of the most relevant mobile agent technologies and their main features. In this section are presented several of the most known MASs which are bound to a single PL, and several others which support many PLs.

6.2.1 Single Programming Language MAS

Grasshopper [BBCM00] is a commercial AM, no longer available, based on the Java PL. It was the first one to support the OMG MASIF and the IEEE-FIPA agent standards. Agents are implemented overriding one of the two super classes provided, the StationaryAgent and the MobileAgent, depending on the type of agent required. These classes provide the methods with the agent and AM functionality, and are implemented by the agent and AM developers respectively. The AM has a communication service with support for CORBA IIOP, Java RMI, and plain socket connections with optional SSL. It also incorporates a persistence service, and a security service which takes advantage of x509 certificates, SSL and the Java security model.

The *Aglets* server middleware [LM98] is an environment to execute Aglets, which are Java MAs. Although its authors considered the Aglet API as lightweight, it is in fact quite large because of the AM extensive functionality, such as the agent creation, migration, cloning, synchronous and asynchronous messaging, and persistence. The agent server is based on an event-driven model. And the interaction with other agents is accomplished by means of a proxy interface, which provides a common and secure way of accessing them.

Ajanta [TKA+02] is an AM for Java MAs which is focused on security and fault tolerance. It supports RMI communications and on-demand agent mobility. Agents are implemented from a base class that comprises several methods which implement the functionality of the agent and the AM. Some of them are overridden by the agent developer to react to agent events and encode its functionality. Agents have the ability to create and kill another agents. Furthermore, Ajanta agents support itineraries including several migration patterns. A specific task can be associated to each visited location.

SeMoA [RJS01] is a runtime environment for Java-based MAs focused on security and easy extensibility. SeMoA agents only have to implement the Java Runnable interface. Services are accessed through context objects got from a static method of the service class. The advantage of this mechanism is that agents only need to know the interfaces of the services they are planning to use.

SOMA [BCS01b, BCS01a] is a flexible and extensible programming framework for MAs with a rich infrastructure of middleware services. It supports local communication by using shared objects, and remote communication by the exchange of asynchronous messages. SOMA includes a persistence service, a QoS service, and a security service, based on PKIs, certificates, and the Java security mechanisms. Furthermore, MASIF and the IEEE-FIPA agent standards are optionally supported.

Tracy [BR05] is a modular, component-oriented, extensible Java AM designed as a micro-kernel. Because of their micro-kernel philosophy their agents only need to implement the Java Runnable interface. Services are accessed by using context objects, each one with its own public interface, which can be retrieved from static classes provided by them. Regarding the life cycle of its agents it only comprises two states: running and waiting.

JADE [BCG06, BCPR08] is a Java AM compliant with the IEEE-FIPA specifications. It has a large interface that provides a complex agent task scheduler, an agent state management completely based on the use of multiple methods (*doSuspend(), doWait()...*), and an agent event management also based on the overloading of several methods (*afterMove(), takeDown()...*). Agent states are based on the IEEE-FIPA agent life cycle [FIP04].

Mobile-C [CCP06] is an IEEE-FIPA compliant AM which supports C/C++ agents and it is based on the extensively use of XML [CLC08]. Although supporting the IEEE-FIPA specifications, the agent life cycle state is proprietary. Its agent interface provides methods to deal with services, communicate, and manage the agent state among others. Agent migrations are done according to a preestablished itinerary. Agents run on top of the Ch C/C++ interpreter.

6.2.2 Multiple Programming Language MAS

TACOMA [JLvR+02] is a framework which supports the execution of MAs developed in different PLs (C, Tcl/Tk, Perl, Python, and Scheme) by using different VMs. Each VM has an associated briefcase folder with the code of the agents running on it. Only weak migration is supported and the agent developer is responsible for gathering the agent data considered necessary in each migration. The communication model of this AM is completely different from the others. It is based on the use of briefcases and cabinets. Briefcases are data containers associated to a specific agent which can be eventually shared with another agent to exchange information. Remote communication is carried out by sending a copy of the agent to the remote platform and exchanging briefcases. Therefore, it implies agent mobility. Cabinets are also data containers, but associated to a specific AP. They are used to concurrently share data among the agents local to the platform. Some of them can be secret by the assignment of large random names only known by a few agents.

D'Agents [GCK+02] is an AM which supports multiple PLs (Tcl, Java, and Scheme). It is composed of a server component and several EEs. Strong migration is supported. Each EE includes an interpreter or VM for the PL considered, a module to capture the agent state, and a module to deal with agent security. There is a set of stub routines to

allow agents to invoke the functions of the AM shared library. This routines are stated in an interface shared by all the agents which keeps the same functionality for each supported PL. Minor differences exist in some method names and parameters defined in the specific interface used in each PL. The server component, which is the core of the AM, manages the reception of agents injecting them to the appropriate EE. Each agent is authenticated and limited to a specific amount of resources. For the sake of simplicity TCL and Scheme EEs are threads within the server process, and the Java EE has its own process.

Ara [Pei02] is an AM which supports different PLs (currently they are Tcl, C/C++, and Java) and strong migration. It is composed of a PL independent system core and several PL interpreters (EEs). The system core provides the minimum low level necessary functionality for agents. Extended functionality must be provided by dedicated server agents (similarly to the IEEE-FIPA philosophy). New EEs can be easily added since there is a well defined interface between the system core and them. The interface is composed of *stubs*, which are the calling interfaces of the core for each supported PL, and the *upcalls*, which are the implementation of certain functions for the interpreters management. Each agent is attached to a system thread. Despite supporting remote communications, local agent communications usage is emphasised. Two methods, one based on synchronous message passing and another based on a shared tuple space, are provided. Ara also provides security by means of an access rights vector mechanism.

AgentScape [OB04] is a multi-language AM designed to support scalable, secure, distributed multi-agent applications. Currently it provides an EE for Java agents. It has a large interface mainly because of the messaging system and the methods to interact with the underlying Agent Operating System (AOS) [vNOT+07]. AOS is a low level layer which provides interoperability and security facilities to AMs based on it. Another AM based on AOS is Mansion [vNBT04].

6.3 Common Agent Interface (CAI)

The existence of different AMs makes the agent interoperability difficult at the middleware level. Each AM has its own agent interface, therefore the interaction agent/AM is limited to entities sharing the same interface. The main contribution of this section is the definition of a common agent interface called CAI within the context of the IEEE-FIPA specifications. A shared specific interpreted PL in each AM is assumed, which currently is the most common scenario in the literature [MR00, PR02, GGK+02, MPD+02, FGR07]. In our opinion, most of the existing full interoperability approaches usually fail in that:

1. *Widespread agent standards, such as MASIF or the IEEE-FIPA specifications, are not taken into account*. It is an important decision whether to use existing agent standards or not in the design of a new agent interface. According to our point of view, not using them is a serious error since it hinders the adoption of the solution proposed and, also, the adoption of the agent standards. Two main agent standardisation efforts stand up, MASIF and the IEEE-FIPA specifications (see Section 2.4.2). MASIF is composed of several definitions and common middleware interfaces (MAFFinder and MAFAgentSystem) for MAS. Nevertheless, no agent standard interface neither agent communication mechanisms are defined. IEEE-FIPA are the standards selected for our approach, since they are the most extended agent interoperability solution nowadays. They provide good interoperability in the area of agent communication, but they do not define a common agent interface. This is the reason why CAI is defined in this section.

2. *Approaches are tightly coupled to a specific PL*. Usually they only support the Java PL and its VM. This is an important limitation, since different PLs are appropriate for different applications. This is the reason why the proposed general interface must be the base of specific interfaces for each PL. This first approach must guarantee interoperability with AMs that share the same PL.

3. *Complex implementations and major middleware internal modifications are required*. Most of the existing approaches are complex and are implemented at

the middleware level, except JIMAF [FGR07] which is implemented at the application level. Many agent interfaces tend to provide unnecessary functionality [GTA08]. The proposed simple agent interface, only with the IEEE-FIPA functionality needed, must be simple. The scheme reinforces the IEEE-FIPA philosophy of using services or doing tasks in cooperation with other agents only by means of their messaging system and not relying on complex architecture dependent interfaces. Furthermore, the implementation must be able to be deployed at the application level or at the middleware level regarding the specific necessities in each context and the availability of the AM source code.

Our proposal is devised for IEEE-FIPA compliant AMs. Nevertheless, non compliant MASs can always be adapted [GRK03, COO+07]. This approach allows the use of different specialised AMs for specific environments such as hospitals, and laboratories among others, which admit generic agents, such as agents based on CAI, to coexist and interact with their native agents by means of the IEEE-FIPA communication facilities. Furthermore, IPMA, presented in Chapter 3, is chosen as interoperability solution in the area of mobility. The reason is that it is a flexible application level agent migration architecture independent of any AM and PL, since it is based on the IEEE-FIPA specifications.

6.3.1 Considerations toward a common agent interface

The life of an agent takes place on top of a specific AM. Due to the fact that there are many different AMs with different agent interfaces, there is no possibility of exchanging their agents. CAI, which is shared by all the AMs with the same PL and UA, allows the exchange of agents compliant with this interface.

The multi-language programming suitability desired for the proposal enforces the definition of an abstract interface as a basis for specific interfaces for any PL (see Figure 6.1). The interface is composed of two parts, one with methods implemented by the agent, which are the ones called by the AM, and another with methods implemented by the AM, which are used by the agent to access the middleware functionality. How this interface is mapped to each PL is freely left to people in charge of the standardisation

regarding each PL. No rules are enforced to allow getting the most appropriate specific interface for each case.

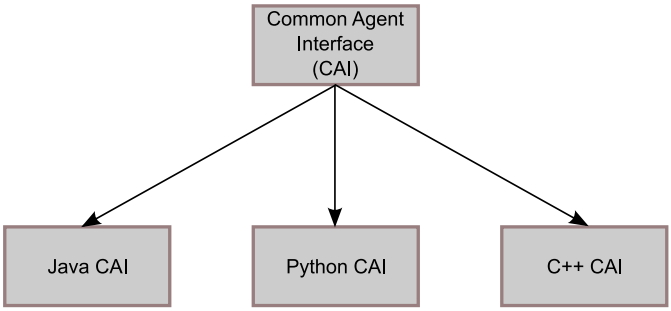

Figure 6.1: Common Agent Interface.

Nevertheless, one may question which is the reason to establish a base interface where specific ones inherit from it if, in fact, the agent interoperability seems to be restricted to systems sharing the same PL and UA. There are two reasons:

- In all the cases the functionality to provide is similar, a set of primitives in the IEEE-FIPA context. Therefore, it is reasonable to use similar interfaces avoiding the effort of designing a new interface for each PL.
- In Section 6.5 there is a proposal of inter-language interoperability, i.e., agents which can move to AMs that support different PLs. These agents are composed of several equivalent codes developed according to a set of similar guidelines, the equivalent interfaces.

Taking into account the interfaces analysed from other AMs (see Section 6.2) and the IEEE-FIPA specifications, CAI should provide the following functionality:

- **Agent management**: The agent must have an IEEE-FIPA [FIP04] compliant unique identification (AID) and life cycle state associated. Several accessors must be provided to set/get them. No accessors are necessary for the agent mobility, since the IPMA model is completely based on the exchange of Agent Communication Language (ACL) messages.
- **Agent messaging**: The exchange of IEEE-FIPA ACL messages [FIP02e] must be supported. Taking into account the nature of ACL messages, asynchronous

communication is enough. Furthermore, some facilities to operate with ontologies should be provided although not as part of CAI.

- **Agent event reaction**: The agent must have the capacity to react to events such as state changes or incoming messages. Nevertheless, the use of this feature is not mandatory and depends on the agent developer.

Although the optimisation of the agent itineraries can bring a huge improvement to the performance of MAs [GP06], no itinerary management facilities [RB02] are directly provided, as Ajanta and Mobile-C AMs do. The reason is that this functionality can be easily provided as part of the agent code without compromising the agent interoperability. Furthermore, no security facilities are also provided through the interface. Since security is usually related to the agent mobility, IPMA is the component which has to implement it (see Section 7.5). Moreover, as ACL messages are used to interact with IPMA, no special methods in the agent interface are required to negotiate security parameters with it.

6.3.2 Proposed Common Agent Interface (CAI)

One interface, some data structures, and a list of values are proposed for CAI. The interface is expressed in terms of object oriented PLs, following the OMG IDL specification [OMG99].

Interface

The interface is composed of two parts. On the one hand, there is the agent interface, which is composed of a set of methods implemented by the agent developer that characterise the agent functionality. On the other hand, there is the AM interface, which is composed of a set of methods with the AM functionality that are implemented by the AM developer. This is the same philosophy that is followed in Monads [MR00] and JIMAF [FGR07] interoperability approaches to avoid mixing the implementation of the two kind of methods. The agent interface is detailed in the following lines:

```
interface Agent {
  void deliver(in ACL msg);
  void setAgentState(in short state);
  short getAgentState();
  void setAID(in AID id);
  AID getAID();
  void run();
  void setMContext(in MiddlewareContext mc);
  MiddlewareContext getMContext();
}
```

The MiddlewareContext object referred by the agent interface is the mechanism the agent might use to access the AM interface, which provides the methods implemented by the AM developer. Therefore, each AM is responsible for assigning its own implementation of this interface using the "`setMContext()`" method. The middleware interface is defined in the following lines:

```
interface MiddlewareContext {
  void send(in ACL msg);
  AID getAMS();
}
```

The interface proposed has a reduced set of methods since only the basic functionality required is provided. The combination of these methods allows developers to implement the exact required functionality for each agent. A clear example is the agent message management, which can be as much complex as developers prefer. They can implement the "`deliver()`" method to manage incoming messages in queues (as the implementation described in Section 3.7.2), or process them instantaneously, or in whatever other way they consider appropriate. Other complex functionality, such as messaging encoding is not included in the interface. Instead of this, this functionality is delegated to the AM, which according to the language and encoding parameters selected by the agent encodes the ACL message.

Data structures

There are two main data structures. They represent the IEEE-FIPA Agent Identifier (AID) [FIP04] and the IEEE-FIPA Agent Communication Language (ACL) [FIP02e] message. First of all the AID:

```
valuetype AID {
  string name
  sequence <string> addresses
  sequence <AID> resolvers
}
```

And the ACL message:

```
valuetype Tuple {
  string key
  string value
}

typedef sequence <Tuple> Table;

valuetype ACL {
  string performative
  AID sender
  sequence <AID> receivers
  AID replyto
  sequence <octet> binary_content
  string string_content
  string language
  string encoding
  string ontology
  string protocol
```

```
  string conversation-id
  string reply-with
  string in-reply-to
  string reply-by
  Table ud-parameters
  string acl-encoding
}
```

Although an agent could use its own data structures to represent this information, the proposed data structures are needed to exchange the information with the AM through the interfaces previously defined. An example are the ACL messages, which are not encoded by the agent, but it is the AM that interprets the data structure previously defined and encodes it according to the language and encoding parameters selected by the agent. Therefore, the message must be handed in to the AM with the data structure defined.

Agent state values

In CAI, a set of integer values are associated to the IEEE-FIPA agent life cycle states (see Section 2.4.3):

```
0 - Unknown
1 - Initiated
2 - Active
3 - Suspended
4 - Waiting
5 - Transit
```

Although the IEEE-FIPA agent life cycle states could be encoded as strings, an integer representation is more efficient and easy to use in the agent code.

6.3.3 Comments on interface usage

CAI is devised to offer the maximum functionality and flexibility to agent developers by providing a set of minimum essential methods. No attributes are enforced, giving the agent developer the freedom to use its own ones. The following lines explain the capabilities of the interface.

Agent management

The AID is managed by the corresponding accessors, "`setAID()`" and "`getAID()`". The AID does not change during the whole life of the agent, except in case of an agent cloning. IEEE-FIPA states the agent identifier must be unique, but does not define mechanisms to enforce it. A possibility, which does not require a central authority, is using a large random value together with the home agent platform's name. The "`getAMS()`" method call allows to get the local Agent Management System (AMS) identification in addition to the network addresses available in the current location. AMS is the agent which, according to the IEEE-FIPA specifications, manages the AP. The agent life cycle state is also managed with similar accessors: `getAgentState()`" and "`setAgentState()`". Notice that in our approach it is not kept by the AMS agent as stated by the IEEE-FIPA Agent Management Specification [FIP04]. This is because the interface proposed is devised for MAs which self-manage their state along their itineraries, i.e., the set of locations visited during the life of an agent. The agent execution is carried out by repeated calls to the "`run()`" method while the agent is in the IEEE-FIPA Active state. The agent is killed by sending a deregistration message to AMS, which changes the agent to the IEEE-FIPA Unknown state (according to IEEE-FIPA only the AMS agent can finally kill agents). No methods are provided to create or destroy agents. This functionality could be provided in a nearby future by means of specifically defined ACL messages sent to the AMS agent. Similarly, services offered by other agents are used through the incorporated messaging facilities. Since the agent migration is considered a service, it is also managed through ACL messages (see Chapter 3) sent to the Agent Mobility Manager (AMM) agent.

Agent messaging

Agent messaging is one of the most flexible parts of CAI. Only two methods are mandatory, one to send messages and another one to receive them (this is similar to the Grasshopper [BBCM00] implementation). The "`send()`" method which is implemented by the AM, directly sends the message passed by parameter. The other one, the "`deliver()`", is used by the AM to deliver an incoming message to the agent. How agents manage these messages is left to agent developers. It is suggested to implement incoming and outgoing message queues and several methods to filter and get messages from them (receive, blockingReceive, and so forth). Notice that no support is provided to deal with ontologies for the ACL message content. This is completely left open, since the establishment of a standard way to deal with them, from the agent interface point of view, valid for any PL is too complex. Two solutions are proposed. On the one hand agents can include all the logic needed to create and parse their messages content. Or, on the other hand, a common library to create and parse ontologies for each PL must be used, e.g., the fiParse [SKW05] generic parser for the Java PL, which is a library included in the AM to encode and parse the message content.

Agent events

There are two basic agent events to deal with, the ones coming from agent state changes, and the ones coming from the messaging system. The first ones can be easily dealt with overriding the "`setAgentState()`" method and taking into account the agent state modification. This can be used, for example, to do some operations just before the agent migrates (the state changes from Active to Transit), or just when it reaches a new location (it changes from Transit to Active) among others. This functionality is offered in some AMs through specific methods overloaded by the agent developer, such as *beforeMove()*, *afterMove()*, *setup()*, *cleanUp()*, and so forth. The messaging system events, i.e., the reception of messages, can be dealt with overriding the "`deliver()`" method, since it is asynchronously called each time a message is received.

6.3.4 Common Agent Interface considerations

The main characteristics of CAI, together with a comparison of regular agent interfaces present in several AMs, are summarised in Table 6.1. Next lines show the most relevant characteristics of it.

First of all, although CAI may seem a single interface, in fact, in Table 6.1, it is classified as two, since one part is implemented by the agent developer and the other by the AM developer. This is a common methodology in interoperability approaches based on interface standardisation (see Monads [MR00] and JIMAF [FGR07] in Section 6.6). Furthermore, it is important to mention that CAI allows an event-driven implementation of the agent, allowing it to react to its state changes according to the agent developer aims. Regarding the message transport and the agent life cycle, they are based on the IEEE-FIPA specifications, therefore they use the FIPA Message Transport Protocols (MTPs) and the FIPA agent life cycle respectively. Regarding the agent creation, disposal and mobility, CAI does not implement any specific method for them, since they can be accomplished by using specific ACL messages defined in the FIPA Agent Management Specification [FIP04] and in IPMA (Section 3). As it can be seen the interface proposed is characterised by its simplicity and flexibility which makes it suitable for different agent models and applications. Furthermore, as stated in [Gav04], having a simple interface, i.e., simple agent classes, a reduced number of mandatory variables, and so forth, contributes to a better performance in agent migrations.

Finally, it is worth mentioning that two realisations of CAI for the Java PL and the Python PL have been developed. They are available in the Mobile Agent Interoperability section of the JIPMS SourceForge project [JIPa]. The two CAI realisations for the Java and Python PLs are also included in Appendix B.

Interface	**Grass-hopper**	**Aglets**	**Ajanta**	**SeMoA**	**SOMA**	**Tracy**	**JADE**	**Mobile-C**	**Tacoma**	**D'Agents**	**Ara**	**Agent Scape**	**CAI**
Number of Interfaces	1 + 1	2	1	n	1	n	1	1	1	1	1	1	2
Enforce agent structure	Yes	Yes	Yes	No	Yes	No	Yes	No	No	No	No	Yes	Yes
Event-driven	No	Yes	Yes	No	No	No	Yes	No	No	No	No	No	Optional
State change reaction	No	No	Yes	No	No	No	Yes	Yes	No	No	Yes	No	Optional
Extensible	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	ACL
Agent creation support	N.A.	Yes	Yes	No	N.A.	No	No	Yes	Yes	Yes	Yes	Yes	ACL
Agent disposal support	N.A.	Yes	Yes	No	N.A.	No	No	Yes	No	No	Yes	Yes	ACL
Agent state management	N.A.	Yes	No	No	N.A.	No	Yes	Yes	No	No	Yes	No	Yes
Mobility support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	ACL
Other facilities	Persis-tence, Security		Security, Fault toler-ance		Persis-tence, QoS, Secu-rity	Place, Sur-vival		Persis-tence			Persis-tence, Security		
Agent standards	MASIF, FIPA	MASIF		FIPA ACL, KQML	MASIF, FIPA		FIPA	FIPA				AOS	Yes
Messaging transport	CORBA IIOP, RMI, sock-ets, FIPA-MTP IIOP	ATP	RMI	Propri-etary	Shared objects, FIPA-MTP, CORBA-IIOP	Propri-etary	FIPA-MTP	FIPA MTP-HTTP	Briefcase	Propri-etary	Propri-etary	AOS (RPC)	FIPA-MTP
Messaging type	Async., Syn.	Async., Syn.	Syn.	Async.	Async., Syn	Async.	Async.	Async.	Async., Syn.	Async.	Async.	Async.	Async.
Messaging locality	Remote	Remote	Remote	Remote	Remote, Local	Local	Remote	Remote	Remote, Local	Remote	Remote, Local	Remote	Remote
Agent lifecycle states	FIPA + Native	2 states		6 states	N.A.	2 states	FIPA	6 states	2 states	2 states	3 states	2 states	FIPA
Programming languages	Java	Java	Java	Java	Java	Java	Java	C/C++	Any	Any	Any	Java and Others	Any

Table 6.1: MAS agent interfaces comparison.

6.4 Multiple execution environments

Interoperability in the area of programming languages and underlying architecture is not completely solved. This area comprises two well differentiated parts. The PL in which an agent is developed and the UA for which it is compiled. The main contribution of this section is proposing an infrastructure to allow agents developed in different interpreted PLs run in the same AP.

6.4.1 Computer architectures

Having multiple computer architectures is a problem currently dealt with the use of interpreted PLs such as Java, and Python among others. In this case a common UA, the one implemented by the VM, is assumed. Other techniques such as reverse engineering to recompile the agents or emulators have been proposed in the literature, although with less acceptation. There is also an attempt to deal with this issue in [OdGWB06] where agent codes are dynamically generated in each visited location. Unfortunately this approach requires too much effort for the AM developers since a set of specific modules of code present in each MAS must be kept updated.

If only one computer architecture existed, there would not be any problem with the PL used in the agent, but to share a set of common dynamic link libraries. In this case agents would be created and distributed compiled for the specific computer architecture with independence of the high level PL used. Since this is not possible, interpreted PL, each one with its own associated UA (interpreter or VM), are used. This is the reason why the solutions proposed in the chapter talk about agents developed in different PLs instead of talking about agents compiled for specific UAs.

6.4.2 A Multiple Execution Environment approach

The approach proposed is based on the use of EEs. An EE is a container of agents which supports a specific PL and agent interface. The EE includes the language interpreter in which their agents are developed and takes care of their execution. One or more EEs can be added to new or existing AMs. Each AP has the more appropriate set of EEs

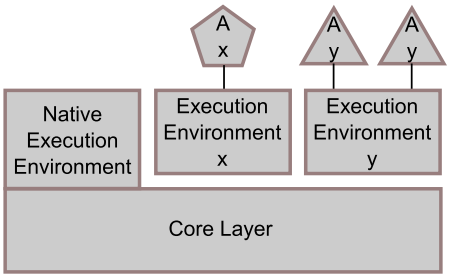

Figure 6.2: Middleware Level.

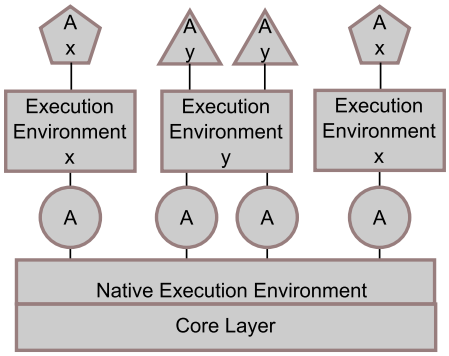

Figure 6.3: Application Level.

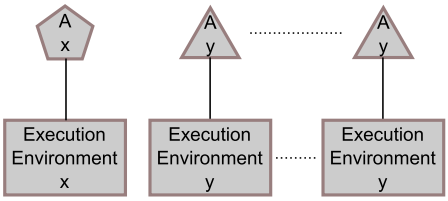

Figure 6.4: 1 Agent.

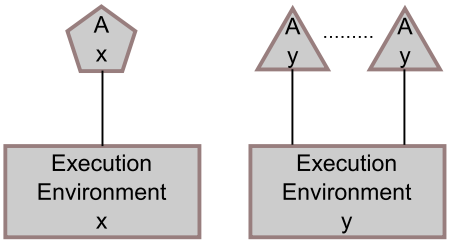

Figure 6.5: N Agents.

considering the expected visitors. Similar approaches are found in Tacoma, D'Agents, and Ara (see Section 6.2.2).

The use of multiple EEs allow the maximum level of rapprochement between agents coded in different PLs. Again the idea of agents interacting locally by means of the IEEE-FIPA Message Transport System (ACL message exchange) is stressed. Nevertheless, in this case agents may be developed with different PLs and are running in different EEs.

In the following lines several alternatives of implementation are explained, all of them compatible from the agent point of view, i.e., the same agent compliant with CAI proposed in the previous section could be executed on each variant. Each alternative is defined by the level on which it is implemented, and the number of agents that each EE hosts. These alternatives are illustrated in Figures 6.2, 6.3, 6.4, and 6.5, where each agent developed with a specific PL is represented with a specific shape and letter (*x* or *y*). The levels on which the EEs and their management components can be implemented are:

- **Middleware level**: This is the most common approach for AMs initially created

with the idea of supporting several PLs (see Figure 6.2), e.g., D'Agents [GCK+02] and Ara [Pei02], or AMs which require an interpret for the PL used, e.g., Mobile-C [CCP06]. In this case agents are directly created within the corresponding EE. The use of a common interface between the core layer of the AM and the EE is highly encouraged although not mandatory to ease the addition of new EEs [Pei02]. Nevertheless, the really important interface for the agent interoperability is the one between the agent and the EE, i.e., the interface proposed in Section 6.3.

- **Application level**: This is the most common approach in interoperability proposals over existing AMs (see Figure 6.3), e.g., Monads [MR00], GMAS [GGK+02], Guest [MPD+02], and JIMAF [FGR07], since it involves a few or no modifications to the AM internals. Because of this, each agent running within an EE is usually represented by a native agent of the AM. This approach usually consumes more system resources than the previous one.

Regarding the number of agents that an EE can host, two variants of the approach are possible:

- **Single agent**: In this case there is only one interoperable agent running on each EE (see Figure 6.4). The simplicity of implementation and the facility to securely isolate agents from other agents (sand boxing) are two of its main advantages. Combined with the application level approach, each EE is running over a middleware native agent. The functionality of the EE is similar to the functionality of an adaptor and, therefore, only one execution thread is required for each agent. An example of its use in the application level, only for the Tcl and Scheme PLs, is the D'Agents [GCK+02] AM. There is also the example of the GMAS [GGK+02] interoperability approach. And an example of its use in the middleware level is the Mobile-C [CCP06] AM.

- **Multiple agents**: In this case there are several interoperable agents running in each EE (see Figure 6.5). This approach is complex to implement, although when it is used at the middleware level is the most efficient one. At the contrary, at

the application level there must be a native agent linked to each interoperable agent running in the external EE, and at least two threads are required for each interoperable agent (see the triangular agents of Figure 6.3). Some examples of its use at the middleware level are D'Agents [GCK+02], regarding the Java PL, and Ara [Pei02], regarding any PL.

All the approaches and their combinations are compatible with CAI. The adoption of one or other method only affects the performance of interoperable agents execution and the complexity of implementation. From now on it is assumed a middleware level approach combined with EEs hosting more than one agent (see it on Figure 6.6).

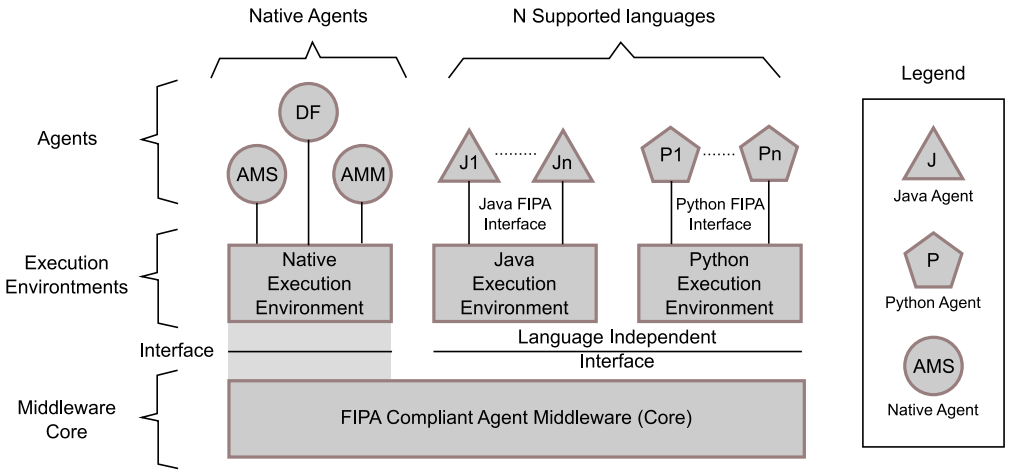

Figure 6.6: Multiple Execution Environments Middleware.

The reasons for choosing this alternative are that having one EE for each agent is quite inefficient in terms of performance execution and resources consumed, and that the implementation at the middleware level is a more robust and well designed option than the implementation at the application level.

In the example of Figure 6.6 there is an AM following this approach with EEs for the Java and Python PLs. In the figure there is an agent called AMM which, as defined in the IPMA mobility specification of Chapter 3, is in charge of agent migrations. See Section 6.5.4 for an explanation of several mobility considerations motivated by the introduction of multiple EEs. Also, notice the existence of the standard AMS and DF IEEE-FIPA agents. As it is explained in Section 2.4.3, AMS is the agent which manages

the agent platform and DF is a directory service.

6.5 Multiple Code Agents

In the previous sections two methods to improve the compatibility of agents with different AMs have been proposed. On the one hand, CAI, which allows agents to live in different types of AM. On the other hand, the multiple EE solution, which increases the range of supported PLs in a single AM. The effectiveness of the first method depends on the support for the PL in which the agent is developed in each of the locations to visit. Therefore, locations should support as many PLs as possible.

The contribution of this section is an additional approach which relaxes this statement. The interface of Section 6.3 is also used. It consists of the creation of agents composed of several equivalent codes developed in different PLs. Therefore, each time an agent visits a new location the appropriate agent code according to the available EEs is selected. This is called from now own inter-language interoperability, since the agent can be transparently executed and migrated through APs which do not support the same PL.

There are two reasons which can hinder the adoption of this approach. Firstly, the agent developer has to provide several codes with equivalent behaviours, which may be a complex task. And secondly, a mechanism to represent the agent data in a standard way must be provided in all the AMs. Otherwise the agent data could not be shared with the different agent codes. In Figure 6.7 there is represented an hypothetical multi-code agent that moves from a Java AM to a Python AM.

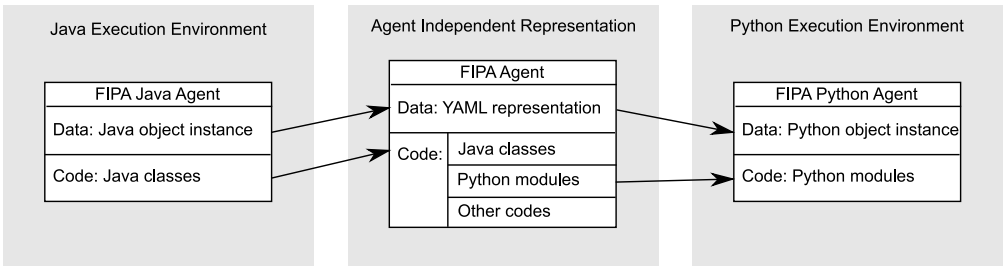

Figure 6.7: Agent data transformation.

Next sections discuss the usage and implementation of agents with multiple codes,

and the standard representation of the agent data by using specific standard data representation languages.

6.5.1 Agent data processing

As previously mentioned, the inter-language compatibility of agents requires a standard mechanism to represent the agent data. In this section a mechanism based on the YAML [yama] data serialisation standard is proposed.

In recent MASs, the methods to capture the agent data have not been object of much discussion. Most of the PLs recently used in agent development are object oriented, and they already include their own object serialisation or marshalling mechanisms. In MASs where agents are represented as objects this is equivalent to capture the agent data. Examples of these mechanisms are the Java serialisation mechanism, and the Python marshalling mechanism, among others. Nevertheless, if compatibility beyond a specific PL is required, the way the agent data is captured and represented is a matter of concern. Therefore, the use of a standard data representation language is recommended.

Existing data serialisation mechanisms

A short analysis of data representation mechanisms is done in the next lines. Two types exist: data binary, and data string. Data binary representation mechanisms are computationally efficient and achieve a good data size ratio, but are not human readable, and are usually tightly coupled to a specific PL, e.g., Java serialisation, Python marshalling, and so forth. Furthermore, sometimes they present class version compatibility issues because non necessary internal information, which is highly version dependent, is captured. Data string representation mechanisms are human readable, but have a worst data size ratio. Nevertheless, since they are not usually tightly coupled to a specific PL they are preferred for our purposes. Some standard data string representation mechanisms have been considered:

- **Extensible Markup Language (XML)**: XML [XML] is a specification which allows the representation of structured data. There is no standard XML data serialisation mechanism defined, but some specific proprietary implementations

with their own semantics for several PLs. The data got from these approaches can be relatively independent of their PLs, and does not present class version compatibility issues like their binary counterparts. Three examples of them are XStream [XSt], which is available only for the Java PL, PyXML [PyX], only available for the Python PL, and .NET XML serialisation [Mic08]. The first one produces a clear XML and allows changes to adjust its dependence to the Java class names (some aliases may be used for them), the second one has a trickier semantic format and is less flexible, and the third one does not support the serialisation of private fields. Nevertheless, there is also an alternative to the automatic object serialisation. It consists of defining a specific data structure where the agent manually stores the data it considers necessary for the next location. This is the alternative used in MAWS [AK06], which is a MAS approach based on Web services.

- **JavaScript Object Notation (JSON)**: JSON [Crob] is a lightweight language independent data-interchange format. It is a subset of the ECMA-262 standard, on which JavaScript is based. Its availability in many PLs, its suitability to encapsulate information, and its reasonable data size ratio, despite being a string based format, are some of its advantages. Nevertheless, since some data types, data references, and cycles are not supported, JSON is not feasible as a general agent data serialisation mechanism.

- **YAML Ain't Markup Language (YAML)**: YAML [yama] is a data serialisation language independent of any specific PL. Its availability in many PLs (although not so extended as JSON), its suitability to encapsulate information, its reasonable data size ratio, its support for generic objects of any data type, and its support for data references and cycles, make YAML the perfect language to store the agent data in a standard way. Several implementations are available for different PLs, e.g., PyYAML [PyY] for Python, YAML.rb [YAMb] for Ruby (which is currently integrated as part of the PL) and Jvyaml [Jyy] for Java.

```
!TestYAMLAgent
acl: !ACL
  encoding: String
  language: SL
  ontology: migration-ontology
  performative: request
  receivers:
  - !AID
    addresses: ['http://localhost:7788/acc',
                'tftp://localhost:7789/acc']
    name: otherAgent@FIPA-Platform/FIPA-P
    resolvers: []
  string_content: content-test
aid: !AID
  addresses: ['http://localhost:7788/acc']
  name: testerAgent@FIPA-Platform/FIPA-P
  resolvers: []
firsthop: 1
state: 4
```

Figure 6.8: YAML encoded agent data.

6.5.2 YAML agent data representation

Our approach is to use the YAML standard data representation language to establish the basis for a standard agent data representation. Some considerations must be taken into account because of the differences in PLs:

- There is a set of basic data types supported by YAML. Nevertheless, specific PL data types and user-defined objects are not part of the standard. If inter-language interoperability is needed, the use of too specific data types is not encouraged and user-defined object classes must be accompanied by a description regarding how to serialise/deserialise them to/from YAML. Usually the only issue with user-defined objects is the name which maps the serialised object to its class. YAML typically names a user-defined object with a pattern which includes the PL name, the class package or module, and the class name, e.g., a Python ACLMessage object in the “fipa” module would be: “!!python/object:fipa.ACLMessage”. Using the methods provided in the YAML implementation of each PL, object class names must be replaced to a user-defined name for each shared object in the different codes provided with the agent, e.g., the ACLMessage object can be identified

as "!ACLMessage". The name assigned must be unique within the agent code. The only needed information to serialise the agent is self-contained in each of the included codes.

- Attribute types are not equally distinguished in all PLs. In some cases types are indicated by the attribute name, e.g., Python private attribute names must begin with "__" characters, which in another language may not be necessary. Therefore, the use of attributes containing PL special characters is discouraged. Proper serialisation/deserialisation descriptions must be provided to map different names to an intermediate valid representation for all the PLs used.

- The default procedure in the serialisation of objects is the capture of all the attributes that belong to the main agent class. Nevertheless, sometimes there are attributes which are not necessary in the remote location, or which can cause difficulties to the serialisation process. As in the other cases this can be adjusted by using the facilities provided by each YAML implementation available for each PL. Including only the minimal required agent state information is a good practise to improve the agent migration performance [Gav04, GTA08].

Following the explained approach with the mentioned considerations, the agent data can be serialised in an independent representation understandable by any of the equivalent agent codes. An example of YAML code produced by the serialisation of an agent, which includes an ACL message, an AID, the IEEE-FIPA agent life cycle state and a user-defined state is shown in Figure 6.8. This is in the context of a prototype available in the MAI package of the JIPMS SourceForge project [JIPa]

6.5.3 Agents with multiple codes

The inter-language interoperability approach proposed in this section is based on the development of several agent equivalent codes, the execution of which can be stopped from one code and resumed to another. Therefore, this approach allows the agent to visit AMs that support different PLs. All agent codes should be designed with an equivalent structure, and they must use the mechanism to capture and restore the agent data

```
def run(self)

  if self.state == 0:
    print "Starting agent
        execution."

    # Do some process
    doWhatEver()

    # Request to migrate
    doMigrate()

    # Setting the next state
    self.state = 1

    print "Ending first hop"


  if self.state == 1:

    print "Ending agent
        execution."

    # Kill the agent
    doKill()
```

```
public void run() {

  switch (state) {

    case 0:
      System.out.println("
          Starting agent
          execution.");

      // Do some process
      doWhatEver();

      // Request to migrate
      doMigrate();

      // Setting the next state
      state = 1

      System.out.println("Ending
          first hop");
      break;

    case 1:

      System.out.println("Ending
          agent execution.");

      // Kill the agent
      doKill();

}
```

Figure 6.9: Equivalent agent codes (Python on the left, Java on the right).

previously discussed. An example of two equivalent codes can be seen in Figure 6.9. Notice that some methods are not present in CAI, since they are implemented by the agent using the more basic CAI methods.

Although these requirements could seem too strong, actually they are not. Equivalent codes do not imply an implementation of exactly the same methods with the same lines of code and the same structure. In fact the only requirement is that all the agent codes deal with the same attributes and have a similar agent life cycle (this last referring to the agent self-defined states in which the code is organised, e.g., the cases of the switch statement in the main agent code if this agent development philosophy is

followed).

But beyond the development of equivalent codes for the inter-language interoperability, new interesting possibilities appear when multiple codes are introduced in agents:

- *Preferred code selection*: When there are several compatible EEs with the agent codes provided, different codes can be chosen when the agent reaches the AP. Therefore, with the appropriate mechanisms, agents and their middlewares can establish a preference in the use of their codes, e.g., to prioritise more efficient PLs such as C++ or similar.

- *Complementary codes*: If the agent developer knows that certain types of tasks are only carried out by AMs developed in a specific PL, it can provide equivalent codes in structure, but only with the needed parts implemented. Supposing an agent with methods A and B, where A is only used in C++ AMs and B is only used in Java AMs, there is no need to implement A in the Java code and vice versa. This is advantageous, since some tasks may be difficult to develop in all the PLs (some PLs are more appropriate than others for specific tasks).

The disadvantages of this approach are the complexity growth in the development of agents, and the increase of the migration time because of the number of codes that must be currently transfered with the agent. Nevertheless, in lots of cases it is worth using the multiple agent code approach because of the previously mentioned advantages.

6.5.4 Agent mobility model considerations

The IPMA mobility model described in Chapter 3 is based on an agent called AMM present in all the AMs which manages agent migrations. The whole migration process is driven by the local AMM, which interacts with the remote AMM until the finalisation of the process. The introduction of the inter-language interoperability requires some additional features to the original AMM, which must be able to:

- manage multiple equivalent agent codes developed in several PLs;

- deal with multiple EEs;

- inject agents in each corresponding EE;
- request YAML agent data capture to an EE;
- control agent execution state through AMS;

Some aspects of the IPMA specification, which in the initial proposals [CMR+07, COO+07] were not considered, have been reinforced to deal with multiple agent codes and different agent data representations:

- Multiple profiles for each agent, referring to each code using a unique code identifier (CID), are supported.
- Agent profiles are included in the preference order of usage. Therefore the agent could establish a specific code preference usage just in case the agent visits some AMs with support for several PLs. If the destination AP does not support any of the agent codes, the migration is cancelled. Then, the agent can decide a new location to visit.
- The transfer of multiple codes for a specific agent must be allowed. This is easy to achieve by creating new transfer protocols to the architecture or by modifying the existing ones. Currently multiple codes can be transfered by using the service described in Chapter 5.
- The representation used for the agent data is specified together with the agent profiles (YAML, XML, Java serialisation...).

Finally, its worth mentioning that in case a secure transmission of the agent data is required, this must be requested to the mobility service (IPMA).

6.5.5 Inter-language mobility example

This section shows the example (see Figure 6.10) of an agent migration in the context of two different AMs taking advantage of our full interoperability proposal. In this case, the EEs are implemented at the middleware level and run several agents. There is an agent A1 that migrates from an AM with a Java EE to an AM with a Python EE. In

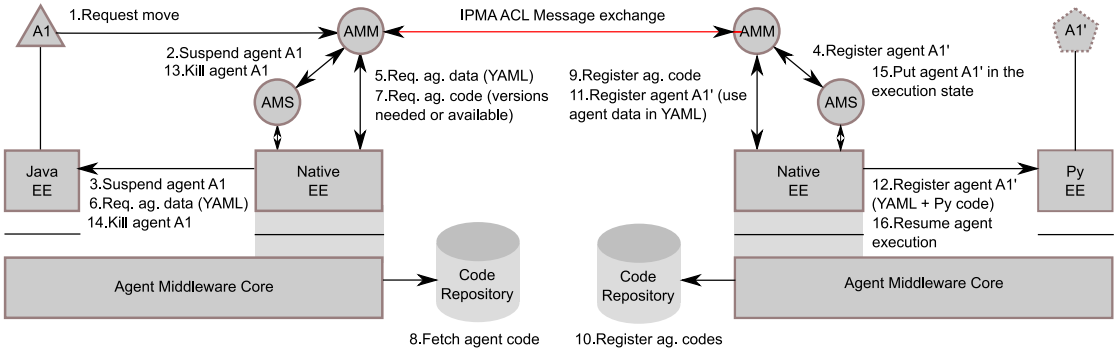

Figure 6.10: Agent migration with YAML encoded data.

the following lines there are detailed the steps of a migration like that, which denote the importance of the different interoperability areas involved in the process.

1. An agent sends an ACL message to AMM to request the movement of itself or, in other cases, the movement of another agent.

2. AMM requests AMS to suspend the migrating agent execution.

3. AMS requests the corresponding EE to suspend the agent.

4. After the first messages defined in IPMA are exchanged, the new agent is registered to the remote AMS.

5. Then the local AMM requests the serialisation of the agent data to the AM.

6. Since the migrating agent does not reside in the native EE, the request to serialise the agent data is forwarded to the Java EE.

7. Therefore, AMM requests the agent code to the AM.

8. In this case the agent code is managed through a local repository. Therefore the codes are requested to it.

9. In the destination, once the codes are received, the agent codes are registered.

10. The AM registers the agent codes to the local repository.

11. Therefore, the remote AMM registers the migrating agent with the YAML agent data previously received.

12. Since in this case the AM only supports the Python PL, the Python agent code is selected. Then, the agent YAML state and the Python agent code are sent to the Python EE.

13. In the local AP, AMM requests AMS to kill the migrating agent.

14. The AMS agent requests the corresponding EE to kill the agent.

15. Finally the remote AMM requests AMS to put the agent in the execution state.

16. The remote AMS resumes the agent execution.

The complete mechanism for the depicted inter-language interoperability approach is composed of an IEEE-FIPA compliant AM that includes several EEs, YAML parser/encoders suitable with each of the in-use PLs, and the IPMA mobility model.

6.6 Related Work

This section compares full interoperability approaches present in the literature with our solutions. Most of the approaches are based on the standardisation of interfaces, universal AMs, agent interface adaptation, and agent regeneration (see Figures 6.11, 6.12, 6.13 and 6.14 respectively). A summary of the comparison can be found in Table 6.2.

Monads [MR00] is a MAS implemented on top of other MASs using the Java PL. The idea behind it is to define a specific interface for its agents which is used to separate Monads agents from the underlying AM. This separation is undertaken by dividing agents into a head, the AP independent part, and a body, the dependent part. In Monads the agent migration and communication is coordinated through a specific agent called *Monads Agent Gateway*. This approach is restricted to the Java PL and, because of the time when it was devised, it is not based on any agent standard (despite including a first implementation of the ACL messages defined by IEEE-FIPA). This proposal is similar to *Guest* from Magnin *et al.* [MPD+02], which runs on top of Java MASs following

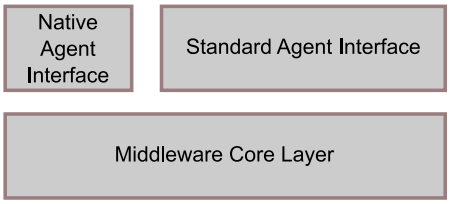

Figure 6.11: Standard Standardardisation.

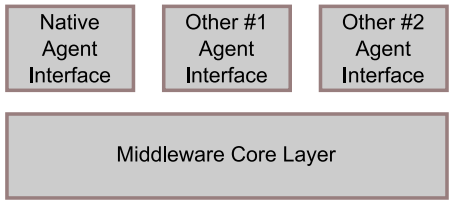

Figure 6.12: Universal Middleware.

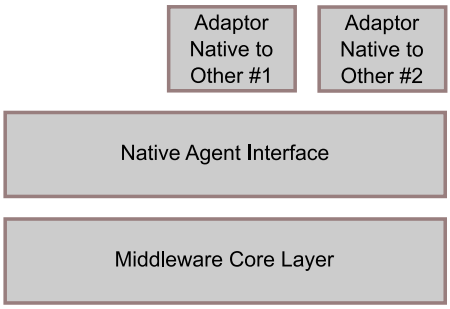

Figure 6.13: Agent Interface Adaptation.

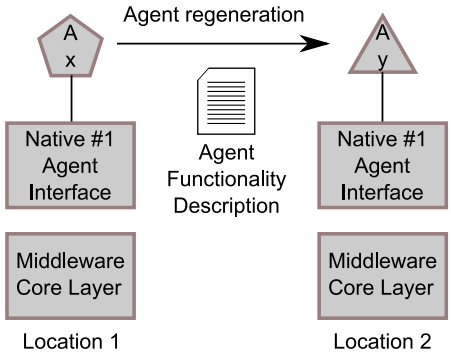

Figure 6.14: Agent Regeneration.

the Guest interface. The approach presented in this chapter is similar to these ones, it is also based on a common agent interface, and the whole migration process is driven by the specific agent AMM. Nevertheless, at the contrary of Monads and Guest, our approach supports different PLs by using multiple EEs, and provides inter-language interoperability by providing several equivalent codes and defining a standard agent data representation.

Pinsdorf *et al.* [PR02] propose a solution integrated into their AM, SeMoA [RJS01], based on voluntary interoperability. SeMoA provides support for interfaces used in other AMs, such as JADE and Tracy, to deal with their agents. In fact, this is an expensive solution, in terms of maintenance costs, and it is not general enough to be extended, since only a subset of AMs can be supported. As new AM interfaces or newer versions of them appear, all the AMs which want to offer compatibility with them must be updated. Furthermore, only the Java PL and its data serialisation mechanism is supported. In SOMA [BCS01b] a similar solution to support agents from the Grasshopper [BBCM00] AM has been applied, although in this case the MASIF standards are also used. The approaches proposed in this chapter do not require such

amount of maintenance, support several PLs, and define a standard agent data representation. Nevertheless, there is the inconvenient that the interoperable agents must be specifically created.

Grid Mobile Agent System (GMAS) [GGK+02] is an approach that allows the execution of native agents of a specific AM to a different one. It relies on the translation of agent interfaces. The idea is to provide an intermediate interface supported by different MASs on top of which are put the translators to the foreign agent interfaces. Then, foreign agents are run above these translators. The advantage of this idea is that agent compatibility can be granted only by creating one GMAS compliant translator which is automatically retrieved when it is needed. Regarding the communications, the CoABS Grid is used to handle them. Nevertheless, GMAS has two main disadvantages, on the one hand it is quite inefficient in performance terms, and on the other hand it is restricted to the Java PL. Proposals present in this chapter can be more efficient and support several PLs. Nevertheless, interoperable agents must be specifically developed following the appropriate interfaces.

Overeinder *et al.* [OdGWB06] is an original proposal that guarantees agent interoperability even in case of different AMs running on different PLs and UA. The approach is based on the regeneration of the agent code each time the agent migrates. The agent's functionality is described by means of the agent's blueprint, an implementation independent description which states a set of components used in the final agent code. These components are implemented in each AM according to its architecture and are used every time the agent code is regenerated. This approach has the advantage that their agents are not constrained to a set of specific PLs and UAs. Nevertheless, the expressivity of the possible agent codes is limited to the components and their implementations available for each possible PL and UA. Furthermore, no explicit solutions are provided regarding the agent communication. From the interoperability point of view, this is the most general approach, since agent developers do not have to provide several equivalent agent codes in different PLs, and AM developers only have to implement a set of modules. Our approach is limited in the number of PL and UA combinations, but it offers more flexibility to agents, since they are not subject to the availability of a set of prebuilt modules.

Java-based Interoperable Mobile Agent Framework (JIMAF) [FGR07] is a high-level approach based on voluntary interoperability. The basic idea is similar to Monads, in the sense that the agent is divided into an AM dependent and an AM independent parts. Interoperable agents must be developed according to a specific interface. The main advantage of this approach is that it does not require the modification of the AM internals, but it is implemented on top of it. JIMAF is composed of several interfaces and several wrapper agents that implement the agent communication and migration facilities. Each JIMAF agent runs on top of a middleware native agent which is an adaptor to the specific JIMAF interface. JIMAF agents are based on a generic light-weight event model. There are two outstanding drawbacks. First of all, the approach is tightly coupled to the Java PL. Secondly, it depends on the use of wrapper agents to interact with native agents. The approach presented in this chapter supports different PLs (even inter-language interoperability), different implementation possibilities (middleware and application levels), and it is based on well-known agent standards which allow the direct interaction with middleware native agents. Furthermore, the proposed agent interface is flexible enough to allow agent developers to follow an event driven model for their agents.

Interoperability solution	**Monads**	**Guest**	**SeMoA**	**GMAS**	**Agent Regeneration**	**JIMAF**	**FIPA Interoperab.**
Interoperability method	Standard interface	Standard interface	Support for interfaces of other MASs	Interface adaptors	Agent regeneration	Standard interface	Standard interface + Execution environments
Agent interface / toolkits	AgentBody / AgentHead	Guest API	Runnable, JADE, Tracy	Any	Agent blueprint	JIMAF	CAI
Inter-language interoperability support	No	No	No	No	Yes	No	Yes
Programming languages supported	Java	Java	Java	Java	Any	Java	Any
Communication interoperability support	Yes	No	Yes	Yes	No	Yes	Yes
Communication solution	Monads Agent Gateway		Wrapping	CoABS Grid		Wrapper agents	FIPA ACL
Migration interoperability support	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Migration solution	Monads Agent Gateway	N.A.	SeMoA migration	GMAS	Agent Factory	JIMAF	IPMA
Agent Data/State representation mechanism	Java Serialization	Java Serialization	Java Serialization	Java Serialization / SelfSerializable agents	XML, RDF or OIL	Java Serialization	YAML
Interoperability transparency	No	No	Yes	Yes	No	No	No
Implementation level	Application level (middleware parts may be required)	Application level	Middleware level	Application level	Application level	Application level	Application / Middleware levels

Table 6.2: Interoperability solutions comparison.

6.7 Conclusions

During the last years a large amount of different MASs have been developed. As a consequence the interoperability of MAs has become a subject of major concern, specially taking into account the present tendency to technological convergence.

In this chapter a full interoperability proposal, which covers all the interoperability areas mentioned in Section 2.4.1, is presented as a combination of the IEEE-FIPA specifications, the IPMA mobility model (see Chapter 3), and the solutions proposed in Sections 6.3, 6.4, and 6.5. The novelties of the proposal, regarding the existing ones described and compared in Section 6.6, are that it is valid for any object oriented PL (and other ones with slight modifications), it is completely integrated with the IEEE-FIPA agent standards, it provides inter-language interoperability, and it defines a common way of saving the agent data using the YAML standard.

The simple agent interface, proposed in Section 6.3, provides the minimal necessary functionality to develop a fully functional agent in an IEEE-FIPA context. The deployment of this approach is simple, in many cases it only consists of an extra layer on top of the AM. Furthermore, since the interface is devised to coexist with the native AM interfaces, the interoperable agents can closely interact with the middleware native agents, which usually have more specific features and properties. Nevertheless, if this approach is the only one used, the interoperability is only guaranteed with agents and AMs that share the same PL. An issue with the implementation of this approach is the management of the agent messages content. A content language is used to represent the information according to a specific ontology. AMs usually provide tools to create and interpret it. Nevertheless, no specific codec/parsers are defined in our approach. This functionality must be implemented in the agent's code or must be obtained from shared libraries. In this last case, a de facto ontology manager must be agreed for each PL.

The concept of EE, discussed in Section 6.4, allows AMs to closely host agents developed in different PLs. The advantage of using EEs is their implementation flexibility. They can be implemented at the middleware level or the application level as appropriate. Nevertheless, a disadvantage is that an EE must be developed for each PL supported by the AM. Therefore, only a limited number of PLs can be supported.

Finally, the method to develop inter-language compatible agents, proposed in Section 6.5, allows agents to run into EEs that support different PLs. The advantage of this approach is that the AM developers do not have to provide different EEs, although in this case the agents must be composed of multiple codes. Another advantage is that having agents with multiple codes may bring the possibility of implementing each agent task with the most appropriate PL, always assuming the agent developer knows in advance the supported PL in each AP. An inconvenient is the agent development complexity increases, although using development automatisation tools it can be greatly reduced. Furthermore, in case of using this approach to implement each task with a specific PL the development complexity can be extremely reduced, since some tasks can be easier to implement with a PL than another. As in the previous approach, a limited number of PLs are supported, although in this case this is limited by the agent codes available. Nevertheless, the two approaches can be combined and the PLs supported increase. There is an issue regarding the implementation of the agent data processing similar to the message content issue previously explained. The fact is that a shared YAML parser must be available in all the AMs for each PL used, otherwise agent codes could not specify how to serialise their data. And, finally, as mentioned in Section 6.5.3, the agent migration time increases as a consequence of the greater number of agent codes to transfer. Nonetheless, using the agent code distribution service presented in Chapter 5, this additional load does not suppose an issue, since the code transport task is delegated to the mentioned service and only the specific code needed in each platform is requested. Otherwise, all the agent code versions must be carried during the whole agent itinerary.

Chapter 7

Security

Security is one of the most trying aspects in Mobile Agent Systems (MASs). In this chapter, after a brief introduction to several security mechanisms, two methods to protect mobile agents are proposed. Firstly, a scheme to protect agent itineraries with loops is presented. And later, a protocol to authenticate agents and guarantee their code's integrity in the Inter-Platform Mobility Architecture (IPMA) is proposed.

7.1 Introduction

Mobile Agents (MAs) introduce new challenges and opportunities for mischief, which must be addressed before we can see real-world applications deployed using them. Security is one of the most trying aspects in MASs, and ensuring sound protection against external malicious parties in MA environments is still an open problem. More concretely, a MA owner must have the guarantee that the agent's code and data will be executed without external modifications. On the other hand, the execution environment must be protected from malicious agents. A set of basic agent security requirements and the most frequent security threats have been explained in Section 2.5.

This chapter begins showing some methods to deal with agent security attacks. Taking into account who the attacker and the victim are and according to [JK00], four categories of attack can be established. The first one is *Agent-to-platform*, which refers

to the set of attacks where one or more agents exploit the Agent Platform (AP) weaknesses. The next one is the *Agent-to-agent*, where agents attack other agents. In this case it must be taken into account that some APs, such as the FIPA compliant ones, have several components operated as agents. Therefore this category might include *Agent-to-platform* attacks. The next one is the *Platform-to-agent*, where APs attack agents. Usually each AP attacks its own agents since they are under its control. This is an attack that it is not easy to prevent because of the agent complete submission to the AP. And, finally, there is the *Other-to-platform* category, where third parties, such as APs or agents, attack an AP. In Section 7.2 several security approaches classified according to the attacked entity (agent or AP) are detailed.

The rest of the chapter is focused on the proposal of two agent security schemes, one for agent itineraries, and the other for IPMA, presented in Chapter 3. The first scheme is an extension of the itinerary protection scheme approach presented in [MB03] and described in Section 7.3. The aim of our proposal is to allow the protection of agent itineraries with loops, which are not supported in the previous approach. This extension is described in Section 7.4 and it has been published in [CAB05, CAOR+05]. Nevertheless, it had been discontinued by the thesis' author, although continued within our research group (see [GMB+08]). The second scheme, described in Section 7.5, is an authentication protocol for MAs devised for the pre-transfer step of IPMA. This protocol checks the authenticity and integrity of the agent codes received in the migration. Finally, the chapter is concluded in Section 7.6.

7.2 Background

In MASs the entities exposed to attacks are agents and APs. Therefore, agent security mechanisms can be classified according to the receiver of the attack.

7.2.1 Agent platforms

The protection of APs is important since agents dwell within them. Therefore, an attack to an AP can indirectly affect a huge amount of agents. Even agent's data may be in danger. APs are easier to protect than agents, since they fit to the conventions of a

traditional computer program. In the next paragraphs there are described some of the most relevant techniques found in the literature.

Software-Based Fault Isolation [WLAG93] is a method to execute non trusted code developed in unsafe Programming Languages (PLs), such as C or C++. This method consists of executing the code in a separate virtual address space, which has a limited amount of resources assigned and strong restrictions to access the network and local file systems. This technique is known as *sandboxing* and it is integrated into the Java Virtual Machine (VM).

The *safe code interpretation* is a method based on the use of interpreted PLs to develop agents. Therefore, potential harmful instructions can be closely controlled to guarantee the system security. Some examples of the PLs that use this technique are Java and Safe Tcl, in which several APs are based on, e.g., JADE [PBK05] and Agent Tcl [Gra96].

The *cryptographic signature of the agent code* is another technique commonly used. Its use allows APs to verify the ownership of a specific code and its integrity. Depending on the AP security policy and the authority which represents the MA, a certain degree of rights are granted to it. As can be seen in [NA06], access control in MAs is complex. There, some solutions to relax its complexity are proposed.

The *State Appraisal* [FGS96] is a method used in combination with the cryptographic signature of the agent code. It consists of several functions integrated in the agent code which verify the agent is in a coherent and permitted state. Modification of the agent code is detected through its signature, and manipulation of the agent state is detected by these functions.

The *Proof Carrying Code* [NL96] mechanism is a technique based on the provision of several formal proofs, which demonstrate the agent code fits to the security policies required by the visited AP. This technique prevents the execution of unsafe code without using any cryptographic mechanism. Nevertheless, it is not an easy method to put into practise.

Another security mechanism, which is focused on the detection of agents coming from non trusted APs, is the one based on the *itinerary logging* [CGH+95, Ord96, Rot98]. Each visited AP adds a signed entry to the log which indicates its identity.

This log is cryptographically secured to avoid manipulation. Therefore, APs can check the locations visited by an agent before giving it rights to access the AP. The only inconvenient is that the log gets bigger as the agent visits new locations. This method is also used to audit agents as it is explained in the next section.

7.2.2 Mobile agents

The protection of MAs is more complex than the protection of APs. The reasons are that MAs and their code do not statically reside in a specific location, but dynamically change regarding their needs, they accumulate the intermediate results obtained in the previous locations, and they are completely subject to the AP where they are residing. Therefore, it is not possible to prevent an attack from a hosting AP. This is the reason why the detection of attacks is emphasised in the mechanisms presented in the next paragraphs. These mechanisms consider the protection of one or more parts of the agent: agent code, data, state, and itinerary (if appropriate).

The first set of mechanisms deal with the encapsulation of results generated by the agent along their execution and itinerary. The encapsulation may imply data integrity, confidentiality, responsibility, and authenticity. One of the first methods proposed was the *Partial Result Authentication Codes (PRAC)* [Yee99]. It consists of the initial creation of a set of disposable random keys used to encrypt the data generated in each location. The Home Agent Platform (HAP), which keeps a copy of all the keys, is the only entity which can retrieve all the data generated. This method can be deceived if the agent visits the same AP more than once, or if there are colluding APs. Therefore, Karjoth *et al.* [KAG98] proposed a mechanism where each entry included a hash of the previous one and the next location to visit. With the addition of this information it is not possible to exchange results hosted in the middle of the chain. Recently, new vulnerabilities, and their correspondingly solutions [CW02, MS03, YFPD04], have appeared, such as attacks based on collusion or based on cutting out the chain and rebuilding it with forged results.

A second type of security mechanism is based on *agent audits*. This mechanism consists of monitoring and recording all the agent actions to detect possible attacks to the agent at the end of its execution. Some of them are the modification of the agent

itinerary, the environment manipulation to make the agent go to a different location, and the agent kidnapping. Several methods have been proposed. One of them [Rot98] is based on two cooperative agents which monitor each other. This method can be generalised for more than two agents. Later, more advanced proposals based on the same idea have been presented [DE04]. Another approach [Sch97] is based on the use of several replicas of the same agent and the logging of their actions. It is assumed that only a subset of the replicas will be kidnapped or lost in case of attack. The considered final result is the one obtained by most of the replicas executed. Finally, another approach based on agent audit is [Vig97]. It tries to discover the agent manipulation from a set of cryptographic traces, regarding the agent execution, kept in the APs visited by it.

APs can get privileged information of their visitor agents and the tasks they are carrying out. This information may be used against the agent's interest. There is no solution to this problem, since APs must have access to agent codes, but two techniques can contribute to alleviate it. The first technique is *Computing with Encrypted Functions* [ST98], which consists of encrypted functions that return encrypted results without having to decrypt the functions, e.g., a function f is encrypted, $E(f)$, and included in the agent code as a program $P(E(f))$. When the agent reaches its destination, the program $P(E(f))$ is executed and the result $E(f(x))$ is obtained. Later, in the HAP, this result is decrypted, $f(x)$. Notice that the destination AP has no access to the function f. The second technique is based on the *code obfuscation* [Hoh98], which consists of making the code incomprehensible to agent developers. The previous technique also falls in the category of code obfuscation.

Since the previous mechanisms present several restrictions or limitations on their applicability, a different approach based on *secure coprocessors* [Yee94, Kar00] can be followed to prevent this kind of information stealing. Secure coprocessors execute agents or, at least, sensitive operations. They use cryptographic private keys and do not allow to monitor the tasks they are processing.

Finally, another important aspect to protect against manipulation are the agent itineraries (see Section 2.3.3). Cryptographic protocols, such as the ones proposed in [CMS99, KT01, Rot02], take advantage of key pairs to guarantee information confidentiality and

integrity to each visited location. An example which illustrates a possible itinerary protection mechanism is shown in the next equation:

$$I = S_{\mathcal{O}}(E_{k_1}(m_1), E_{k_2}(m_2), ..., E_{k_n}(m_n)) \tag{7.1}$$

this is an agent itinerary (I) composed of a vector signed $S_{\mathcal{O}}$ by the agent owner ($\mathcal{O}$) which contains a set of ciphered tasks ($E_{k_i}(m_i)$). Each task is intended to be executed in a different location, since it is ciphered with a specific key k_i associated to the location. Nevertheless, these proposals are focused on the protection of static itineraries. This is the reason why several proposals [SRM98, MB02, MB03] (see the following Section) make the static itineraries more flexible by adding different types of transitions (sequence, alternative, and set). All these proposals assume security mechanisms only involve the agent and the visited APs, but in [GRCR04, TY05] an alternative method based on Trusted Third-Parties (TTPs) is presented. The disadvantage of this method is the big infrastructure required. More recent works [GRB08] deal with dynamic itineraries, but using only a subset of trusted nodes which belong to the agent itinerary.

7.3 Protection of agent itineraries

In this section a scheme for agent itinerary protection described in [MB03], which is extended in the next section, is presented. That scheme is flexible enough to allow arbitrary combinations of sequences, alternatives, and sets to specify the protected itineraries over a fixed list of hosts. To protect an itinerary, its representation as a Petri net is first constructed. This construction allows an efficient treatment of complex paths, and provides a clear-cut specification of the transitions to be protected.

The protection protocol is based on digital envelopes [Lab93] with the structure:

$$\mathcal{D} = (P_j(r_i)|E_{r_i}(I))$$

where r_i is the (randomly generated) envelope's symmetric key and P_j denotes the destination host's public key. The owner of the corresponding private key can therefore obtain r_i to decipher the information I. The protected itinerary is built as a chain of

digital envelopes, so that it can only be disclosed in a pre-defined order. Let us describe how the protection protocol works.

Protection Protocol:

1 Initialisation : Itineraries are represented as a Petri net with n nodes labelled h_i for $1 \leq i \leq n$. The agent's owner, $\mathcal{O}$, generates n random keys, r_i, and assigns one to each node. Each node represents a host to be visited. We denote $t_{i,j}$ the transition from node i to node j (see transition examples in Figures 7.1 and 7.2).

2 For each transition $h_i \Rightarrow h_j$, create $t_{i,j}$:

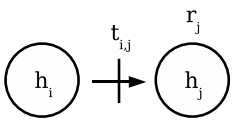

Figure 7.1: Sequence type transition.

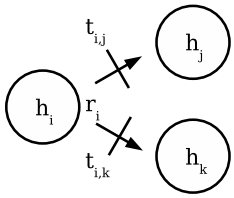

Figure 7.2: Alternative type transition.

(a) Signature: The owner signs the address of h_i and h_j, r_j and a travel mark t (see Equation 7.4 below), obtaining:

$$S = S_{\mathcal{O}}(h_i, h_j, t, r_j)$$

(b) Transition token: The owner ciphers S using h_j's public key; the transition token $t_{i,j}$ is then obtained as the concatenation of this value and h_j address:

$$t_{i,j} = (h_j, P_j(S_{\mathcal{O}}(h_i, h_j, t, r_j))) \tag{7.2}$$

3 Information ciphering: The agent's owner proceeds to encipher each node's private information, including the next transitions, using the corresponding symmetric key r_i. This information will consist of a method or task, m_i and, for alternate entries, a condition, $\mathcal{C}$ (see figure 7.2). If we denote by E_i the operation of enciphering with key r_i we can write the ciphered information e_i, representing an alternative, as:

$$e_i = E_{r_i}(m_i, \mathcal{C}, t_{i,j}, t_{i,k}). \tag{7.3}$$

4 End : Once the above steps have been performed for each node in the Petri net, we end up with a protected itinerary, which can be denoted as $I = (e_1, e_2, \cdots, e_n)$.

The travel mark t identifies the agent, and precludes replay and cut-and-paste attacks. It is defined as:

$$t = T \parallel H(\text{mobile agent}'\text{s code}), \tag{7.4}$$

where T is a time stamp and H a cryptographic hash function.Without this mark, a dishonest host could change the agent's itinerary, using an old agent's one (belonging to the same owner). T guarantees that t is unique for each travel, while the agent's code hash binds the itinerary to its legitimate agent. Each visited host will store t only as long as the agent owner's signatures remain valid, so that the agent's validity is time-limited.

7.3.1 Properties of the Protection Protocol

The protection protocol just described can be applied to itineraries created by arbitrary combinations of sequential and alternate subpaths, and it will provide the following guarantees [MB03]:

P1 Integrity

P2 Confidentiality

P3 Forward privacy

P4 Data verifiability

P5 Originator's verifiability

P6 Strong identification

P7 Once entry

The P6 property, which guarantees that the agent is executed only once in each host, by means of its unique identification, precludes loops in the itinerary. To allow them, we must provide a way of distinguishing legitimate re-executions (ensuing from the agent following a closed path during its travel) from malicious replays. The following sections show how this can be done without jeopardising the protection scheme.

7.4 Protection of agent itineraries with loops

In this section a method to protect agent itineraries with loops, such as the one shown in Figure 7.3, is presented. The method is a modification of the agent itinerary protection protocol described in the previous section.

7.4.1 Protection Protocol Modifications

The existence of closed paths, the loops, will obviously imply that the agent's code is executed more than once in one or more nodes of its itinerary. The challenge is to distinguish such legitimate re-executions from malicious ones issued by an attacker.

Let N denote the vector of maximum allowed agent executions at each itinerary node for any given time; i.e.,

$$N = (seq_1, seq_2, \ldots, seq_n)$$

where seq_i is the maximum number of times the agent can visit host h_i during the rest of its travel. At the origin, this vector will be initialised with the maximum allowed visits to each node, and, each time the agent is executed at host h_i, its corresponding counter seq_i will be decremented. The key of our protocol is to include this vector in the information that travels with the agent. Rather than N, the agent will contain the execution counters in ciphered form, as the vector:

$$v_\omega = (E_{r_1}(seq_1), E_{r_2}(seq_2), \ldots, E_{r_n}(seq_n)), \tag{7.5}$$

where we have made explicit this vector's dependency on ω, the step in the agent travel. That is, at the agent's origin we are at step $\omega = 0$, after visiting the first host in the itinerary, $\omega = 1$, and so forth until the travel ends for some value $\omega = \omega_f$.

E_{r_i} denotes the symmetric ciphering function using key r_i; as mentioned, only the agent's owner and h_i have access to the corresponding visits counter. The former will initialise the counter with the maximum allowable number of visits to the host for a given itinerary, while the latter will take care of checking and decrementing it after each visit.

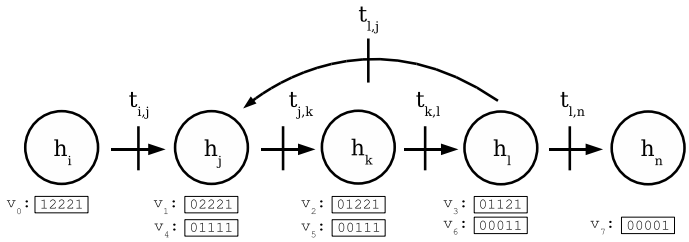

Figure 7.3: One loop itinerary.

The protection provided by the symmetric ciphering of seq_i can be further strengthened by adding to the counter random bits at fixed positions, which will be changed every time seq_i's value is recomputed in order to avoid predictability-based attacks.

In the example depicted in figure 7.3, if we wish the agent visits each internal loop nodes twice, we would have $seq_i = seq_n = 1$ and $seq_j = seq_k = seq_l = 2$, and, during the agent's itinerary, v_ω will take eight different values, i.e., ω will run from 0 to 7.

The first time an agent visits h_i the following steps will ensue:

- The host deciphers $E_{r_i}(seq_i)$ and the travel token t. Since this is the first agent's visit, no record containing t will be found in the host's database.
- A time to live, TTL, is assigned to the above values. This TTL will be greater or equal to the agent's expiration time, so that the agent cannot be re-executed after the stored counters have expired (and possible deleted from the system).
- The tuple
$$t_i = (t, seq_i, \mathrm{TTL}) \tag{7.6}$$
is stored.
- If $seq_i > 0$, the agent is executed, its counter decremented and v_ω updated accordingly.

On the other hand, every time the agent closes a loop and re-visits h_i the following process will take place:

- Extract from the vector v_ω carried by the agent its current execution counter, seq_i.

- If the agent has expired or $seq_i = 0$, the agent will be rejected.
- Retrieve the tuple t_i (see Equation (7.6)) associated with this agent, using t as a search key.
- If agent's execution counter, seq_i, is not lesser than tuple's (t_i) associated counter, the agent is rejected.
- Otherwise, the agent is executed.
- t_i is updated substituting his counter for seq_i's agent counter.
- The agent's data is updated subtracting one unit to seq_i in v_ω, and the agent migrates to its next destination.

A key ingredient of the above protocol is the vector v_ω carried by the agent. Figure 7.4 provides a schematic view of the main agent components. If v_ω, as given by Equation 7.5, would be transported without further protection, a malicious third party could capture it and try to tamper with the system using substitution attacks (see Section 7.4.3).

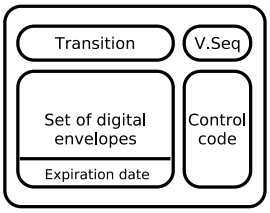

Figure 7.4: Mobile agent components.

To avoid substitution attacks, the execution counter vector will be ciphered so that it can only be read by the next node in the agent's itinerary. In addition, the receiving node should be able to check the legitimacy of the agent's sender (i.e., the previous host in the itinerary, which was the last v_ω modifier). These goals can be met by storing in the agent, instead of v_ω, the following quantity:

$$p_i = P_j(S_i(v_\omega, H(t_{i,j}))), \tag{7.7}$$

where i is the sending host, j the receiver and $t_{i,j}$ the transition given by Equation 7.2. Since the vector is ciphered using the receiver node's public key (this ciphering being denoted by the function $P_j()$ above), only the receiver can access it. The digital signature S_i provides verifiability of p_i's creator identity. Finally, the hash function H binds p_i to the corresponding transition.

This completes our new protocol definition. Note that the creation of p_i should be fully automated, avoiding the agent to directly deal with the private keys of the visited APs. To that end, the cryptographic services scheme proposed in [AROR04] can be used.

7.4.2 Loop Implementation

Our new protocol can be implemented, in a manner analogous to that of alternatives, in systems using itinerary protection mechanisms of the kind described in Section 7.3.

For instance, to implement the loop depicted in Figure 7.3, we just need two additional transitions: one from the final host to the initial host ($t_{l,j}$) and one between the final and next hosts ($t_{l,n}$), which exits the loop. Both transitions will be signed by the agent's owner and ciphered for the corresponding destination node (as described in [MB03]):

$$t_{l,j} = (h_j, P_j(S_O(h_l, h_j, t, r_j))), \tag{7.8}$$

$$t_{l,n} = (h_n, P_n(S_O(h_l, h_n, t, r_n))), \tag{7.9}$$

In addition, these transitions will be contained in the digital envelope assigned to the last node in the loop, with the same format used for an alternative, namely:

$$e_l = E_{r_l}(m_l, \mathcal{C}, t_{l,j}, t_{l,n}), \tag{7.10}$$

where $\mathcal{C}$ denotes the guard condition to be fulfilled for the transition to the initial node to take place.

Finally, the execution counters will be initialised at origin with the corresponding values for the maximum number of allowed executions in each host.

7.4.3 Security Assessment

This section examines our proposed protocol resilience against external attacks.

External Itinerary Replay

In replay attacks, as depicted in Figure 7.5, a malicious external entity captures roaming agents and tries to modify and execute them at hosts on times different to the intended ones. Agents can be captured by network sniffing or collusion with a host in the agent's itinerary.

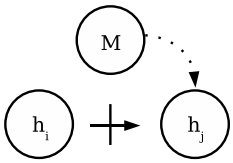

Figure 7.5: External Reply Attack.

The use of a Public Key Infrastructure (PKI) and the associated encryption, and the use of verification services, such as the comparison of the sequence number stored in the AP with the one provided by the agent, precludes this kind of attacks in our protocol. These techniques are used also in the original itinerary protection protocols, so that both the static and dynamic parts (see Figure 7.4) of our agents are protected. As mentioned, the use of public key encryption and digital signatures to protect this data avoids any impersonation risks (a third party trying to play the part of a legitimate host).

Substitution attacks will also be detected. In this case, the attacker would try to send her own agent to resume the captured agent's itinerary. A new execution counter vector could be constructed with the correct structure, but it would nevertheless be detected as bogus when trying to get the agent counters, because symmetric keys will be different.

Internal Itinerary Replay

In this scenario, the attack is conducted by a dishonest host in the agent's itinerary, which tries to re-execute the agent in another node to the host's advantage (e.g., to make

it buy more items than planned at the host's electronic shop). As shown in Figures 7.6 and 7.7 this attack can be attempted either by an isolated host or in collusion with other hosts in the itinerary.

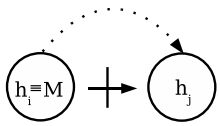

Figure 7.6: Single host internal replay attack.

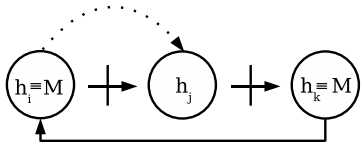

Figure 7.7: Internal replay attack with collusion.

Single host It can be easily seen that this attack has no chance of success. Despite having access to v_ω, the dishonest host cannot alter the execution counter of any other host in the itinerary, for it has no access to the symmetric key of the attacked host.

Colluding hosts In this attack, two dishonest nodes collude to cause unauthorised agent executions on a third honest host (see Figure 7.7). When the MA reaches the second dishonest host, this host sends v_ω to the colluding AP (both of them will have access to its contents, via their private keys). The receiver will inject this new vector in a copy of the agent (which had visited the AP in a previous itinerary step), appropriately signed and ciphered, and it will resend the agent to the attacked host. The latter will have no way of detecting the fraud, and will re-execute the agent. In this way, unintended agent executions can be provoked and go undetected: our protocol cannot cope with collusion.

Internal Host Replay

In this attack, a dishonest host re-executes the agent as many times as it sees fit, disregarding the execution counter. As noted in the literature [TY05], this kind of attack is unavoidable, due to the fact that it is not observable from the outside.

Loop Iteration Replay

A final security breach arises when the host that decides when a loop needs to be iterated is dishonest. Such a host has a privileged role, and can provoke as many iterations of a closed loop as desired, unless the number of iterations is constant and can be fixed beforehand, and the loop is composed of more nodes. In the latter case, the maximum number of executions set at the agent's origin will also be the total number of executions for any legitimate itinerary traversal, and the attack would be detected. In itineraries where the actual number of loop iterations is dynamically determined by the agent, the only solution is to ensure the critical host's honesty by external means.

7.5 IPMA Security Protocol

Agent security solutions presented in previous sections are not bound to a specific AP or agent migration architecture. They are general enough to be implemented in any of them. On the contrary, in this section a security solution regarding the agent authentication specifically designed for IPMA, presented in Chapter 3, is proposed.

This solution is implemented as an IPMA pre-transfer protocol called One-Shot Agent Authentication Protocol (OSAAP). It is an authentication protocol, which is based on the exchange of Agent Communication Language (ACL) messages, that identifies MAs with X.509 certificates [Croa], and that guarantees the ownership and integrity of the agent and their codes using digital signatures. A secure channel to exchange the ACL messages of the whole migration protocol is assumed, therefore no manipulation of the Migration Identifier (MID) is possible.

7.5.1 Preliminaries

Before explaining the protocol there are two kinds of preliminaries to deal with: cryptography, and agent code organisation.

Firstly, the aim of this protocol is to authenticate agents and assure their code integrity and authenticity. This can be achieved by means of the cryptographic signature operation. This operation requires to deal with certificates and signatures. The X.509

standard [Croa] has been chosen to represent the certificate of the agent owner ($\mathcal{O}$). The PKCS-7 cryptographic data structure [Kal] has been chosen to encapsulate the mentioned certificate and the signatures done with it. Furthermore, it is worth noting that required signatures are calculated in advance by the agent owner when the agent is created. Sometimes, new codes can be developed and signed later. No method is enforced to deal with private keys since they are only used by the agent owner ($\mathcal{O}$).

The agent code organisation, see Section 3.3.2, imposes several restrictions to the way agents are authenticated. Since new agent codes can be added to a specific agent after its creation, it is not possible to include all the code signatures in advance with the agent. Therefore, two kinds of signatures are distinguished: the *agent signature*, and the *code signature*. The *agent signature* (see Equation 7.11) is created through the signature of a string containing the agent name (a_{name}) and the Code Group Identifier (CGID), which binds the agent with a group of codes, separated by the colon character. A PKCS-7 structure encapsulates the agent signature and the certificate of the agent owner ($\mathcal{O}$). And the *code signature* (see Equation 7.12) is built by signing a URN, such as the one proposed in Section 5.3.2, which uniquely identifies each code and binds it with its CGID, Code Identifier (CID), Security Revision (SR), and Hash Code Identifier (HCID). It is also encapsulated in a PKCS-7 data structure, although no certificate is included in this case.

Each agent includes an *agent signature*. And each agent code has a *code signature* associated. Agent codes which are obtained from external code distribution services, such as the Agent Code Distribution Service (ACDS) presented in Chapter 5, must be accompanied by a *code signature*. Therefore, new codes compliant with this security mechanism can be created and distributed for already existing and deployed agents.

$$agent\_signature = S_{\mathcal{O}}(a_{name} + CGID) \tag{7.11}$$

$$code\_signature = S_{\mathcal{O}}(code_{urn}) \tag{7.12}$$

$$code_{urn} = urn : agent - code - id :< CGID >:< CID >:< SR >:< HCID > \tag{7.13}$$

7.5.2 Protocol's operation

The authentication process consists of two parts (see Figure 7.8), one carried out in the pre-transfer step and the other one in the transfer step. Firstly, an X.509 certificate identifying the agent owner ($\mathcal{O}$) and a set of signatures are sent. Checking this certificate, and the possible chain of certificates behind it, remote APs decide whether to accept an agent or not depending on the authorities they trust. The agent is bound to the certificate presented by the *agent signature* (see Equation 7.11).

Secondly, OSAAP also takes part indirectly in the transfer step. There, the *code signature* (see Equation 7.12) is validated against the code received. If the code cannot be validated an error is climbed up to the transfer protocol. Although seems that this affects other protocols, since code validation failures are forwarded by the transfer protocols, this is only an implementation challenge. From the transfer protocol point of view this implies sending a normative failure message with the the description returned by the OSAAP implementation. It is also important to take into account that not all the combinations of the authentication and transfer protocols are available. It depends on the developer's decisions and it is announced by the Directory Facilitator (DF) agent (see Section 3.3.2).

The protocol is composed of one FIPA Request interaction protocol, and a specific ontology (see Appendix A.9) that defines the associated action to the request and the mentioned information included in it. Errors are managed in the same way as in MMP (see Section 3.3.3), although with specific exception predicates regarding this protocol. An example of a typical authentication in terms of ACL messages is shown in Figure 7.9.

7.6 Conclusions

Security of MAs is one of the most discussed issues of this technology. If agents cannot offer enough security to their users, then their deployment is difficult. Nevertheless, if they are not enough spread, it is difficult to know which are the most important security problems to deal with. In this chapter a summary and a classification of the most common security problems present in MASs have been shown. Therefore, two specific security issues have been analysed and two solutions have been proposed.

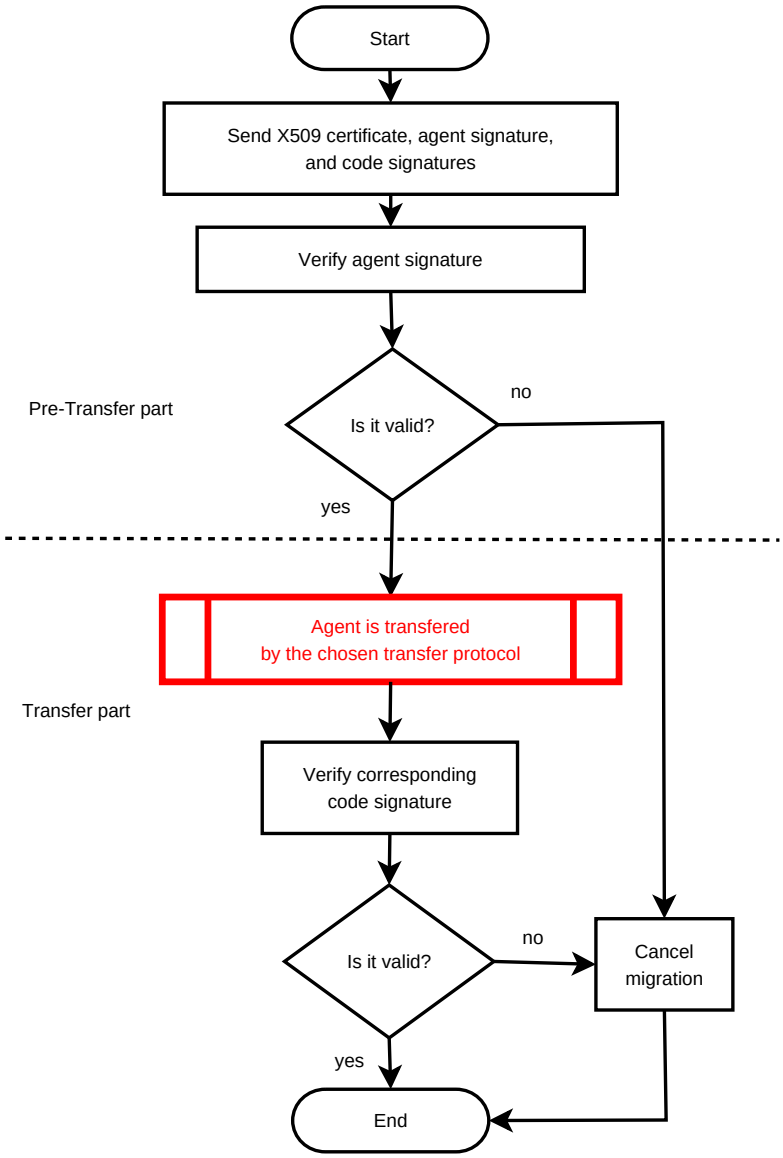

Figure 7.8: One-Shot Agent Authentication Protocol diagram.

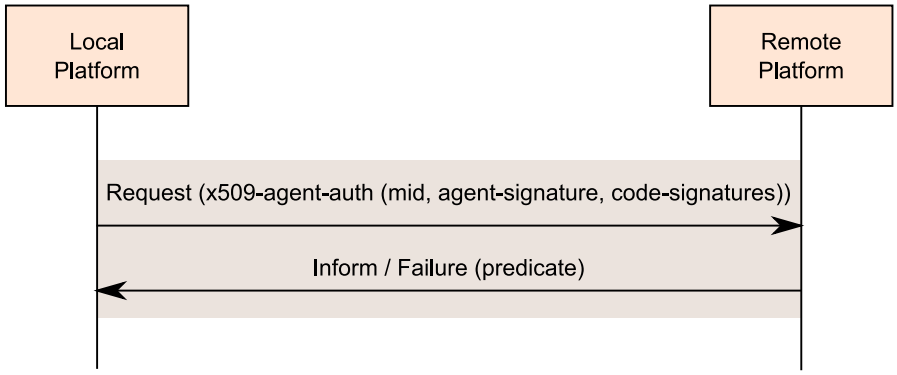

Figure 7.9: One-Shot Agent Authentication Protocol flow diagram.

Firstly, a method [MB03] to protect agent itineraries has been studied. The method fails in protecting the itineraries which contain loops. Therefore, a protocol, which is an improvement of the previous one, to support the existence of loops is presented.

Finally, in the context of IPMA, a new security protocol to authenticate agents and guarantee their code integrity has been devised. This is an example of agent authentication based on X.509 certificates. Certificates identify agent owners, in which agent platforms decide if they trust or not, and a set of digital signatures prove that agents and their codes really belong to their agent owners and they have not been modified.

Chapter 8

Conclusions

As explained in the introduction of this thesis, great progress has been made in the mobile agent technology since its creation. Nevertheless, despite the research efforts, a wide-scale adoption of the technology has not been produced. There are several reasons about that, but the most important one nowadays is focused on the interoperability of Mobile Agents (MAs). Furthermore, the efficiency and security are also important.

In order to deal with the interoperability of mobile and non mobile agents several initiatives arise. The most widespread initiatives are the IEEE-FIPA specifications, which are focused on the management and communication of agents. Nevertheless, they do not take into account their mobility. Another initiative specifically created for MAs is MASIF, but it is not currently in use, and it requires the active collaboration of the Agent Middleware (AM) developers to achieve migration interoperability between different AMs.

The first objective of the thesis has been the design of an agent mobility specification to complement the existing IEEE-FIPA standards. The resulting specification has been described in Chapter 3. It is the Inter-Platform Mobility Architecture (IPMA), an efficient, flexible, and extensible migration architecture which is completely based on the use of the IEEE-FIPA specifications. Several protocols can be incorporated to the mentioned architecture in order to provide customisable agent migrations. Thus, a set of protocols with different migration strategies, i.e., different ways of transferring the agent components, have also been proposed in the same chapter. The research of this

chapter has been published in several conferences [ACM+06, CMR+07, COO+07] and it has been submitted to a journal [CMNA+08]. Furthermore, IPMA has been implemented in two AMs. There is a first implementation, the JIPMS add-on for the JADE AM, which has a project associated to the SourceForge website [JIPa]. And, furthermore, there is the ASIPMS for the AgentScape AM, which can be obtained directly from the author. The first implementation is considered the default inter-platform mobility service of JADE, this is the reason why it motivated a chapter [CAM06] in the book written by the JADE developers. Finally, an extensive performance analysis of the JIPMS implementation with all the protocols proposed has been shown in Chapter 4. The aim of this analysis has been demonstrating the flexibility and efficiency of the architecture presented, this last depending on the migration protocols used.

Another objective of this thesis was the proposal of efficient methods to distribute agent code. Since the agent code is usually static during the whole life of an agent, and sometimes is shared with several agents, having a specific service that deal with it can highly improve the agent migration performance. In Chapter 5 there is a proposal of a distribution service called Agent Code Distribution Service (ACDS). In the literature exists some proposals to distribute agent codes [GGGO02, BR05], but all of them are tightly coupled to a specific type of code (Java code). ACDS is valid for any type of existing or future agent code, introduces the concept of code contract based on XACML to specify the code distribution parameters and restrictions, and have an standard and simple interface based on REST. A migration performance comparison with and without the service enabled has been carried out in the same chapter. Therefore, the improvement that the service proposed implies has been demonstrated. The research of this chapter has been submitted to a journal [CNAM+08]. Furthermore, a prototype [acd] has been implemented.

Coming back to the interoperability of MAs, with the introduction of IPMA described in Chapter 3 it is possible to send MAs between different types of AM. Nevertheless, since no interoperability exists at the middleware level regarding the agent design, it is not possible to execute the incoming agent if it is not compatible with the destination AM. Despite several proposals exist, there is no agreement in the literature to face this problem. The deployment of these proposals is not easy, since they are not

integrated with existing agent standards and, moreover, they are usually focused on only one specific Programming Language (PL).

This is the reason why the third objective of this thesis was proposing methods to guarantee the interoperability of agents at the middleware level, taking into account the possibility of different PLs and Underlying Architectures (UAs). Therefore, in the end, three methods have been proposed in Chapter 6. The combination of them with the IEEE-FIPA standards and IPMA allows agents to move and resume their execution in a location with a different type of AM. Each of the methods proposed is an improvement over the degree of interoperability reached. The first method proposes a common agent interface that provides the agent with all the IEEE-FIPA functionality. The second method defines the concept of execution environment, which is a way to structure AMs in order to cope with agents developed with different PLs and UAs. And the third method propose the creation of agents supporting different PLs and UAs by carrying several versions of their code. These last method also defines a standard way to share the agent data using the YAML language. As this last method can be expensive in terms of the agent code transmission, the ACDS service usage is recommended. The interoperability research detailed in this chapter has been submitted to a journal [CMNA$^+$].

Finally, although it has not been the main objective of the thesis, the security of agents has also been discussed. Two different aspects of the agent security have been dealt with in Chapter 7. First of all, the security of agent itineraries (list of locations that an agent visits). In this part a proposal to protect preestablished agent itineraries that includes loops has been presented. This work starts from a protocol to protect agent itineraries proposed in [MB03]. This research has been published in two conferences [CAB05, CAOR$^+$05]. The second aspect dealt with is the access control of agents and the authentication and integrity of their codes within IPMA. In this case a protocol for the migration architecture has been proposed. This protocol checks, during the migration of an agent, the identity of the agent to decide if it is accepted in the destination. Furthermore it checks the authenticity and integrity of their code. This solution can be combined with ACDS, since this last supports the distribution of the code signatures required to authenticate the agent codes.

Therefore, as a final conclusion it can be stated that the work done during this thesis

allows MAs to efficiently migrate between different types of AM, with different PLs and UAs, and with some security measures.

8.1 Future research lines

The future directions of the research presented in this thesis are detailed in the next paragraphs. First of all, regarding IPMA, a more complete security analysis to guarantee the robustness of the architecture against migration attacks should be considered. And a study of new fault tolerance techniques may also be convenient. Furthermore, new migration protocols to integrate in the architecture can be proposed in the future. These protocols can include security mechanisms for the agent migration, such as the one detailed in Chapter 7.

In ACDS, three ideas can be considered. Firstly, since currently the agent code distribution is only based on the code owner instructions (a list of regions), then a predictive method to distribute the agent code, based on collected statistics of code usage can be studied. Secondly, the service is used through a REST interface, then additional interfaces can be devised to ease developers the service integration into their products. And thirdly, a solution to distribute portable source code and serve it compiled on the fly for the required UAs would allow the deployment of agents developed with highly efficient PLs, such as C or C++, over almost any architecture supported by the service.

Finally, future research in the interoperability must be focused on the deployment of the solutions presented in the whole thesis, such as the combination of the IEEE-FIPA standards together with IPMA and the methods proposed in Chapter 6, in different types of AM with different PLs and UAs. The results and experiences obtained from these deployments will allow a final adjustment of the solutions proposed.

Bibliography

[acd] Agent Code Distribution Service 0.1. http://tao.uab.cat/acds.

[ACM+06] J. Ametller, J. Cucurull, R. Martí, G. Navarro, and S. Robles. Enabling mobile agents interoperability through fipa standards. In M. Klusch, M. Rovatsos, and T.R. Payne, editors, *Cooperative Information Agents X*, volume 4149 of *Lecture Notes in Artificial Intelligence*, pages 388–401, Edinburgh, UK, September 2006. CIA 2006, Springer Verlag.

[AK06] Hassan Artail and Elie Kahale. Maws: a platform-independent framework for mobile agents using web services. *J. Parallel Distrib. Comput.*, 66(3):428–443, 2006.

[App] Appstream. http://www.appstream.com/.

[ARB03] J. Ametller, S. Robles, and J. Borrell. Agent Migration over FIPA ACL Messages. In *Mobile Agents for Telecomunication Applications (MATA*, volume 2881 of *Lecture Notes in Computer Science*, pages 210–219. Springer Verlag, 2003.

[AROR04] J. Ametller, S. Robles, and J. A. Ortega-Ruiz. Self-protected mobile agents. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 362–367, Washington, DC, USA, 2004. IEEE Computer Society.

[AWS] Amazon Web Services. http://aws.amazon.com/resources.

[Bar02] Albert-László Barabási. *Linked: The New Science of Networks*. Perseus Publishing, April 2002.

[BBCM00] C. Baumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper: a universal agent platform based on omg masif and fipa standards, 2000.

[BCG06] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, January 2006.

[BCPR08] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Information and Software Technology*, 50:10–21, 2008.

[BCS01a] P. Bellavista, A. Corradi, and C. Stefanelli. Mobile agent middleware for mobile computing. *Computer*, 34(3):73–81, Mar 2001.

[BCS01b] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. Middleware services for interoperability in open mobile agent systems. *Microprocessors and Microsystems*, 25(2):75–83, April 2001.

[BGN+05] Ch. Bouras, A. Gkamas, I. Nave, D. Primpas, A. Shani, O. Sheory, K. Stamos, and Y. Tzruya. Application on demand system over the internet. *Journal of Network and Computer Applications*, 28(3):209–232, August 2005.

[BLFMa] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform resource identifiers (uri): Generic syntax. http://www.ietf.org/rfc/rfc2396.txt.

[BLFMb] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2617: Http authentication: Basic and digest access authentication. http://www.ietf.org/rfc/rfc2617.txt.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A formalism for specifying multiagent software systems. In *Agent-Oriented*

Software Engineering: First International Workshop, AOSE 2000, Limerick, Ireland, volume 1957 of *LNCS*, pages 109–120. Springer-Verlag, 2001.

[BR05] P. Braun and W. R. Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann, Heidelberg, Germany, 2005.

[CAB05] J. Cucurull, J. Ametller, and J. Borrell. Protocol for the protection of mobile agent itineraries with loops (in Spanish). In Alberto Peinado Domínguez et al., editor, *1r Simposio sobre Seguridad Informática [SSI'2005]*, pages 61–68, Granada. Spain, September 2005. CEDI 2005, Thomson.

[Cai04] G. Caire. Jade: The new kernel and last developments. Technical report, Telecom Italia, 2004. http://jade.tilab.com/papers/Jade-the-services-architecture.pdf.

[CAM06] J. Cucurull, J. Ametller, and R. Martí. Agent mobility. In F. L. Bellifemine, G. Caire, and D. Greenwood, editors, *Developing Multi-Agent Systems with JADE*, pages 115–130. Wiley, January 2006.

[CAOR+05] J. Cucurull, J. Ametller, J.A. Ortega-Ruiz, S. Robles, and J. Borrell. Protecting mobile agent loops. In T. Magendanz, K. Ahmed, and I. Venieris, editors, *Mobility Aware Technologies and Applications*, volume 3744 of *Lecture Notes in Computer Science*, pages 74–83, Montreal, Canada, October 2005. MATA 2005, Springer.

[CCP06] Bo Chen, Harry H. Cheng, and Joe Palen. Mobile-c: a mobile agent platform for mobile c/c++ agents. *Softw., Pract. Exper.*, 36(15):1711–1733, 2006.

[CFL+02] Jiannong Cao, Xinyu Feng, Jian Lu, Henry Chan, and Sajal K. Das. Reliable message delivery for mobile agents: Push or pull. *icpads*, 00:314, 2002.

[CGH+95] Davis Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5):34–49, 1995.

[CGK+05] Krzysztof Chmiel, Maciej Gawinecki, Pawel Kaczmarek, Michal Szymczak, and Marcin Paprzycki. Efficiency of jade agent platform. *Sci. Program.*, 13(2):159–172, 2005.

[CHB03] Arjav J. Chakravarti, Xiaojing Wang; Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in java. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*. IEEE, 2003.

[CLC08] Bo Chen, David D. Linz, and Harry H. Cheng. XML-based agent communication, migration and computation in mobile agent systems. *Journal of Systems and Software*, 91(9):1364–1376, 2008.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java*, Manchester (UK), April 2000.

[CMNA+] J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, and J. Borrell. Full mobile agent interoperability in an IEEE-FIPA context. Submitted to Journal of Systems and Software.

[CMNA+08] J. Cucurull, R. Martí, G. Navarro-Arribas, S. Robles, B. J. Overeinder, and J. Borrell. Agent mobility architecture based on IEEE-FIPA standards. 2008. Submitted to Computer Communications.

[CMR+07] J. Cucurull, R. Martí, S. Robles, J. Borrell, and G. Navarro. FIPA-based interoperable agent mobility. In *Multi-Agent Systems and Applications V*, volume 4696 of *LNAI*, pages 319–321, Leipzig, Germany, September 2007. Springer.

[CMS99] A. Corradi, R. Montanari, and C. Stefanelli. Mobile agents protection in the internet environment. In *23rd Annual International Computer Software and Applications Conference*, 1999.

[CNAM+08] J. Cucurull, G. Navarro-Arribas, R. Martí, S. Robles, and J. Borrell. Agent mobility architecture based on IEEE-FIPA standards. 2008. Submitted to Journal of Network and Computer Applications.

[COO+07] J. Cucurull, B. J. Overeinder, M. A. Oey, J. Borrell, and F. M. T. Brazier. Abstract software migration architecture towards agent middleware interoperability. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 27–37, Wisla, Poland, October 2007.

[Croa] D. Crockford. ITU-T Recommendation X.509. Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks.

[Crob] D. Crockford. RFC 4627: The application/json media type for javascript object notation (JSON). http://www.ietf.org/rfc/rfc4627.txt.

[Cuc04] J. Cucurull. JADE MTP-TFTP. Technical report, Universitat Autònoma de Barcelona, June 2004.

[CW02] J.S.L. Cheng and V.K. Wei. Defenses against the truncation of computation results of free-roaming agents. In R. Deng, S. Qing, F. Bao, and J. Zhou, editors, *Information and Communications Security: 4th International Conference, ICICS 2002*, volume Volume 2513 of *Lectures Notes in Computer Science*, pages 1–12. Springer-Verlag GmbH, January 2002.

[DE04] Asnat Dadon-Elichai. Rds: Remote distributed scheme for protecting mobile agents. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 354–361, Washington, DC, USA, 2004. IEEE Computer Society.

[DMP+02] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *Internet Computing, IEEE*, 6(5):50–58, Sep/Oct 2002.

[EARL03] J.A. Exposito, J. Ametller, S. Robles, and N. Lhuillier. How to use the new HTTP MTP with JADE. Technical report, Universitat Autònoma de Barcelona, 2003.

[FGR07] Giancarlo Fortino, Alfredo Garro, and Wilma Russo. Achieving mobile agent systems interoperability through software layering. *Information and Software Technology*, 2007. doi:10.1016/j.infsof.2007.02.016.

[FGS96] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, 1996.

[FIP00] FIPA. FIPA agent management support for mobility specification, 2000. `http://www.fipa.org/specs/fipa00087/index.html`.

[FIP02a] FIPA. FIPA abstract architecture specification, 2002. `http://www.fipa.org/specs/fipa00001/index.html`.

[FIP02b] FIPA. FIPA acl message representation in bit-efficient specification, 2002. `http://www.fipa.org/specs/fipa00069/index.html`.

[FIP02c] FIPA. FIPA acl message representation in string specification, 2002. `http://www.fipa.org/specs/fipa00070/index.html`.

[FIP02d] FIPA. FIPA acl message representation in xml specification, 2002. `http://www.fipa.org/specs/fipa00071/index.html`.

[FIP02e] FIPA. FIPA ACL message structure specification, 2002. `http://www.fipa.org/specs/fipa00061/index.html`.

[FIP02f] FIPA. FIPA agent message transport protocol for http specification, 2002. `http://www.fipa.org/specs/fipa00084/index.html`.

[FIP02g] FIPA. FIPA agent message transport protocol for iiop specification, 2002. `http://www.fipa.org/specs/fipa00075/index.html`.

[FIP02h] FIPA. FIPA agent message transport service specification, 2002. `http://www.fipa.org/specs/fipa00067/index.html`.

[FIP02i] FIPA. FIPA communicative act library specification, 2002. `http://www.fipa.org/specs/fipa00037/index.html`.

[FIP02j] FIPA. FIPA propose interaction protocol specification, 2002. `http://www.fipa.org/specs/fipa00036/index.html`.

[FIP02k] FIPA. FIPA request interaction protocol specification, 2002. `http://www.fipa.org/specs/fipa00026/index.html`.

[FIP02l] FIPA. FIPA SL content language specification, 2002. `http://www.fipa.org/specs/fipa00008/index.html`.

[FIP04] FIPA. FIPA agent management specification. Internet, 2004. http://www.fipa.org/specs/fipa00023/index.html.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, 1998.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Interet Technol.*, 2(2):115–150, 2002.

[Gav04] Damianos Gavalas. Mobile agent platform design optimisations for minimising network overhead and latency in agent migrations. In *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, volume 2, pages 605–609, 2004.

[GCK+02] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'agents: Applications and performance of a mobile-agent system. *Softw., Pract. Exper.*, 32(6):543–573, 2002.

[GGGO02] Damianos Gavalas, Dominic Greenwood, Mohammed Ghanbari, and Mike O'Mahony. Hierarchical network management: a scalable and dynamic mobile agent-based approach. *Computer Networks*, 38(6):693–711, 2002.

[GGK+02] Arne Grimstrup, Robert Gray, David Kotz, Maggie Breedy, Marco Carvalho, Thomas Cowin, and Daria Chacón. Toward interoperability of mobile-agent systems. In *Mobile Agents*, volume 2535 of *LNCS*, pages 106–120. Springer Berlin / Heidelberg, January 2002.

[git] GIT - Fast Version Control System. http://git.or.cz/.

[GMB+08] C. Garrigues, N. Migas, W. Buchanan, S. Robles, and J. Borrell. Protecting mobile agents from external replay attacks. *Journal of Systems and Software*, 2008.

[Gon01] Li Gong. Jxta: a network programming environment. *Internet Computing, IEEE*, 5(3):88–95, May/Jun 2001.

[GP06] Damianos Gavalas and Christina Tanya Politi. Low-cost itineraries for multi-hop agents designed for scalable monitoring of multiple subnets. *Computer Networks*, 50(16):2937–2952, November 2006.

[Gra96] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, Monterey, CA, 1996.

[Gra03] Jim Gray. Distributed computing economics. In Andrew Herbert and Karen Spärck Jones, editors, *Computer Systems: Theory, Technology and Applications*, pages 93–101. Springer, December 2003. Also MSR-TR-2003-24, March 2003.

[Gra04] R.S. Gray. Mobile agents: overcoming early hype and a bad name. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 302–303, 2004.

[GRB08] C. Garrigues, S. Robles, and J. Borrell. Securing dynamic itineraries for mobile agent applications. *Journal of Network and Computer Applications*, 31(4):487–508, November 2008.

[GRCR04] M. Giansiracusa, S. Russell, A. Clark, and V. Roth. Macro-level attention to mobile agent security: Introducing the mobile agent secure hub infrastructure concept. In *Information and Communications Security: 6th International Conference, ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 343–357. Springer Verlag, 2004.

[GRK03] Christos Georgousopoulos, Omer F. Rana, and Anthony Karageorgos. Supporting fipa interoperability for legacy multi-agent systems. In *Proceedings of the Agent-Oriented Software Engineering IV*, number 2935 in LNCS, pages 361–379. Springer, 2003.

[GTA08] Damianos Gavalas, George E. Tsekouras, and Christos Anagnostopoulos. A mobile agent platform for distributed network and systems management. *Journal of Systems and Software*, doi:10.1016/j.jss.2008.06.034, 2008.

[Hem05] Stephen Hemminger. Network emulation with netem. Linux Conf Au, 2005.

[Hoh98] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113, London, UK, 1998. Springer-Verlag.

[HS07] March Hadley and Paul Sandoz. Jax-rs 1.0 early draft review specification, October 2007. Sun Microsystems, Inc.

[HY98] Matthew Hohlfeld and Bennet Yee. How to migrate agents, August 1998. unpublished.

[IKKW01] Torsten Illmann, Tilman Krueger, Frank Kargl, and Michael Weber. Transparent migration of mobile agents using the java platform debugger architecture. In *Mobile Agents: 5th International Conference*, volume 2240 of *Lecture Notes in Computer Science*, page 198. Springer Verlag, December 2001.

[JIPa] JADE Inter-Platform Mobility Service. http://jipms.sourceforge.net.

[JIPb] JADE Inter-Platform Mobility Service Performance Test Suite. http://jipms.sourceforge.net.

[JJK06] Kresimir Jurasovic, Gordan Jezic, and Mario Kusek. A performance analysis of multi-agent systems. *International Transactions on Systems Science and Applications*, 1(4):335–342, 2006.

[JK00] W. Jansen and T. Karygiannis. Nist special publication 800-19 - mobile agent security, 2000.

[JLvR+02] Dag Johansen, Kåre J. Lauvset, Robbert van Renesse, Fred B. Schneider, Nils P. Sudmann, and Kjetil Jacobsen. A tacoma retrospective. *Softw. Pract. Exper.*, 32(6):605–619, 2002.

[Jyy] Jvyaml. https://jvyaml.dev.java.net/.

[KAG98] G. Karjoth, N. Asokan, and C.A. GÃ¼lcÃ¼. Protecting the computation results of free-roaming agents. In *MA '98: Proceedings of the Second International Workshop on Mobile Agents*, pages 195–207, London, UK, 1998. Springer-Verlag.

[Kal] B. Kaliski. RFC 2315: PKCS #7: Cryptographic message syntax. http://www.ietf.org/rfc/rfc2315.txt.

[Kar00] G. Karjoth. Secure mobile agent-based merchant brokering in distributed marketplaces. In D. Katz and F. Mattern, editors, *Second Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA 2000)*, number 1882 in LNCS, pages 44–56. Springer, 2000.

[KE06] Gu Su Kim and Young Ik Eom. Domain-based mobile agent fault-tolerance scheme for home network environments. In *Information Security Practice and Experience, LNCS*, volume 3903 of *LNCS*, pages 269–277, February 2006.

[KT01] Neeran M. Karnik and Anand R. Tripathi. Security in the Ajanta mobile agent system. *Software Practice and Experience*, 31(4):301–329, 2001.

[Lab93] R. Laboratories. PKCS 7: Cryptographic message syntax standard, 1993.

[LCW04] M. R. Lyu, X. Chen, and T. Y. Wong. Design and evaluation of a fault-tolerant mobile-agent system. *IEEE Intelligent Systems*, 19(5):32–38, Sept.-Oct. 2004.

[LM98] Danny B. Lange and Oshima Mitsuru. *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[LRW03] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, pages 112–120, June 2003.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[MB02] J. Mir and J. Borrell. Protecting general flexible itineraries of mobile agents. In *Proceedings of ICISC 2001*, LNCS. Springer Verlag, 2002.

[MB03] J. Mir and J. Borrell. Protecting mobile agent itineraries. In *Mobile Agents for Telecommunication Applications (MATA)*, volume 2881 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, October 2003.

[MFB+07] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications*. Microsoft patterns & practices. Microsoft Press, 2007.

[Mic08] Microsoft. *.NET Framework Developer's Guide*, 2008. XML and SOAP Serialization.

[MMLCVN08] Iván Marsá-Maestre, Miguel A. López-Carmona, Juan R. Velasco, and Andrés Navarro. Mobile agents for service personalization in smart environments. *Journal of Networks*, 3(5):30–41, May 2008.

[MMLVA06] I. Marsá-Maestre, M.A. López, J.R. Velasco, and A.Navarro. Mobile personal agents for smart spaces. In *ACS/IEEE International Conference on Pervasive Services*, pages 299–302, 2006.

[MOB06] D. G. A. Mobach, B. J. Overeinder, and F. M. T. Brazier. WS-Agreement based resource negotiation framework for mobile agents. *Scalable Computing: Practice and Experience*, 7(1):23–36, 2006.

[MPD+02] L. Magnin, T. Viet Pham, A. Dury, N. Besson, and A. Thiefaine. Our guest agents are welcome to your agent platforms. In *Seventeenth ACM Symposium on Applied Computing (SAC)*, pages 107–114, 2002.

[MR00] P. Misikangas and K. Raatikainen. Agent migration between incompatible agent platforms. *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 4–10, 2000.

[MS03] Paolo Maggi and Riccardo Sisto. A configurable mobile agent data protection protocol. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 851–858, New York, NY, USA, 2003. ACM Press.

[MSM97] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, 1997.

[NA06] G. Navarro-Arribas. *Access Control and Authorisation Management in Mobile Agent Systems*. PhD thesis, Universitat Autonoma de Barcelona, 2006.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without runtime checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[NQKA98] Jan Nicklisch, Jürgen Quittek, Andreas Kind, and Shinya Arao. Inca: an agent-based network control architecture. In *IATA '98: Proceedings of the second international workshop on Intelligent agents for telecommunication applications*, pages 142–155, London, UK, 1998. Springer-Verlag.

[OB04] B. J. Overeinder and F. M. T. Brazier. Scalable middleware environment for agent-based Internet applications. In *Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04)*, pages 675–679, Copenhagen, Denmark, June 2004. Published in Applied Parallel Computing, LNCS 3732, Springer, Berlin, 2006.

[OdGWB06] B. J. Overeinder, D. R. A. de Groot, N. J. E. Wijngaards, and F. M. T. Brazier. Generative mobile agent migration in heterogeneous environments. *Scalable Computing: Practice and Experience*, 7(4):89–99, December 2006.

[OMG97] OMG Mobile Agent Systems Interoperability Facilities Specification (MASIF), OMG TC Document ORBOS/97-10-05 , 1997.

[OMG99] OMG. The common object request broker: Architecture and specification. Technical report, OMG, 1999.

[OPB01] James J. Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000, Limerick, Ireland*, volume 1957 of *LNCS*, pages 201–218. Springer-Verlag, 2001.

[Ord96] J. J. Ordille. When agents roam, who can you trust? In *First Conference on Emerging Technologies and Applications in Communications (etaCOM)*, Portland, OR, 1996.

[O'S] Bryan O'Sullivan. Distributed revision control with mercurial.

[PB07] Mukaddim Pathan and Rajkumar Buyya. A taxonomy and survey of content delivery networks. Technical report, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, 2007. GRIDS-TR-2007-4.

[PBK05] D. Trinh P. Braun and R. Kowalczyk. Integrating a new mobility service into the jade agent toolkit. In K. Ahmed T. Magendanz and I. Venieris, editors, *Mobility Aware Technologies and Applications*, volume 3744 of *Lecture Notes in Computer Science*, pages 354–363, Montreal, Canada, October 2005. MATA 2005, Springer.

[Pei02] Holger Peine. Application and programming experience with the ara mobile agent system. *Softw., Pract. Exper.*, 32(6):515–541, 2002.

[PR02] U. Pinsdorf and V. Roth. Mobile Agent Interoperability Patterns and Practice. In *Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 238–244. IEEE Computer Society Press, 2002.

[PyX] Pyxml. http://pyxml.sourceforge.net/.

[PyY] Pyyaml. http://pyyaml.org/.

[RB02] Emmanuel Reuter and FranÃ§oise Baude. System and network management itineraries for mobile agents. In *Mobile Agents for Telecommunication Applications*, volume 2521 of *Lecture Notes in Computer Science*, pages 227–238, 2002.

[RFI02] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[Riv] R. Rivest. RFC 1321: The md5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt.

[RJS01] Volker Roth and Mehrdad Jalali-Sohi. Concepts and architecture of a security-centric mobile agent server. In *ISADS '01: Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems*, page 435, Washington, DC, USA, 2001. IEEE Computer Society.

[RKSB05] Hariharan Rahul, Mangesh Kasbekar, Ramesh Sitaraman, and Arthur Berger. Towards realizing the performance and availability benefits of a global overlay network. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, November 2005.

[Rot98] Volker Roth. Secure recording of itineraries through co-operating agents. In *ECOOP Workshops*, pages 297–298, 1998.

[Rot02] V. Roth. Empowering mobile software agents. In *Proc. 6th IEEE Mobile Agents Conference*, volume 2535 of *Lecture Notes in Computer Science*, pages 47–63. Spinger Verlag, 2002.

[Rot04] V. Roth. Obstacles to the adoption of mobile agents. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 296–297, 2004.

[Rou05] David Roundy. Darcs: distributed version management in haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM.

[Sat03] I. Satoh. Building reusable mobile agents for network management. *Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 33(3):350–357, Aug. 2003.

[SBS00] L.M. Silva, V. Batista, and J.G. Silva. Fault-tolerant execution of mobile agents. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 135 – 143, New York, NY, June 2000.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. Technical report, Cornell University, Ithaca, NY, USA, 1997.

[SKW05] Markus B. Söllner, Sven Kaffille, and Guido Wirtz. fiParse - a generic parser for FIPA-compliant agent communication. In Peter Kokol, editor, *IASTED Conf. on Software Engineering*, pages 331–336. IASTED/ACTA Press, 2005.

[SLBW05] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. Acms: the akamai configuration management system. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 245–258, Berkeley, CA, USA, 2005. USENIX Association.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[Sof] SoftIntegration. Ch interpreter.

[SRM98] M. Straßer, K. Rothermel, and C. Maiï¿½fer. Providing Reliable Agents for Electronic Commerce. In *Proceedings of the International IFIP/GI*

Working Conference, volume 1402 of *Lecture Notes in Computer Science*, pages 241–253. Springer-Verlag, 1998.

[ST98] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, London, UK, 1998. Springer-Verlag.

[TKA+02] Anand R. Tripathi, Neeran M. Karnik, Tanvir Ahmed, Ram D. Singh, Arvind Prakash, Vineet Kakani, Manish K. Vora, and Mukta Pathak. Design of the Ajanta system for mobile agent programming. *Journal of Systems and Software*, 62(2):123–140, 2002.

[TST+05] H.A. Thant, Khaing Moe San, Khin Mar Lar Tun, T.T. Naing, and N. Thein. Mobile agents based load balancing method for parallel applications. *Information and Telecommunication Technologies, 2005. APSITT 2005 Proceedings. 6th Asia-Pacific Symposium on*, pages 77–82, November 2005.

[Tut04] Kurt Tutschku. A measurement-based traffic profile of the edonkey file-sharing service. In *Passive and Active Network Measurement*, volume 3015 of *LNCS*, pages 12–21, 2004.

[TY05] Y. Tsipenyuk and B. Yee. Detecting external agent replay and state modification attacks. http://www-cse.ucsd.edu/˜ytsipeny/home/research/paper.pdf, February 2005.

[VFL06] S. De Capitani Di Vimercati, A. Ferrero, and M. Lazzaroni. Mobile agent technology for remote measurements. *IEEE Transactions on Instrumentation and Measurement*, 55(5):1559–1565, October 2006.

[Vig97] G. Vigna. Protecting mobile agents through tracing. In *Third Workshop on Mobile Object Systems*, 1997.

[Vig04] G. Vigna. Mobile agents: ten reasons for failure. *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 298–299, 2004.

[VMRC+06] P. Vieira-Marques, S. Robles, J. Cucurull, R. Cruz-Correia, G. Navarro, and R. Martí. Secure integration of distributed medical data using mobile agents. *IEEE Intelligent Systems*, 21(6):47–54, 2006.

[vNBT04] G.J. van't Noordende, F.M.T. Brazier, and A.S. Tanenbaum. Security in a mobile agent system. *Multi-Agent Security and Survivability, 2004 IEEE First Symposium on*, pages 35–45, 30-31 Aug. 2004.

[vNOT+07] Guido van't Noordende, Benno Overeinder, Reinier Timmer, Frances M. T. Brazier, and Andrew Tanenbaum. A common base for building secure mobile agent middleware systems. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 13–25, Wisla, Poland, October 2007.

[WHB01] Xiaojin Wang, Jason Hallstrom, and Gerald Baumgartner. Reliability through strong mobility. The Ohio State University, June 2001.

[Whi96] James E. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, Menlo Park, CA, 1996.

[WHN+01] Johnny Wong, Guy Helmer, Venkatraman Naganathan, Sriniwas Polavarapu, Vasant Honavar, and Les Miller. Smart mobile agent facility. *Journal of Systems and Software*, 56:9–22, 2001.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[WPT+97] D. Wong, N. Paciorek, T.Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (Berlin, Germany)*, volume 1219 of *LNCS*. Springer-Verlag, April 1997.

[xac05] eXtensible Access Control Markup Language (XACML) Version 2.0, February 2005. T. Moses (ed.).

[XML] Extensible markup language (xml) 1.0. http://www.w3.org/TR/xml.

[XSt] Xstream. http://xstream.codehaus.org/.

[yama] YAML Ain't Markup Language Version 1.1. http://www.yaml.org/spec/1.1/.

[YAMb] Yaml.rb. http://yaml4r.sourceforge.net/.

[Yee94] Bennet Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.

[Yee99] Bennet S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, pages 261–273, 1999.

[YFPD04] M. Yao, E. Foo, K. Peng, and E. Dawson. An improved forward integrity protocol for mobile agents. In Moti Yung Kijoon Chae, editor, *Information Security Applications*, volume Volume 2908 of *Lectures Notes in Computer Science*, pages 272–285. Springer-Verlag GmbH, January 2004.

Appendix A

Inter-Platform Mobility Architecture

This appendix contains technical details about IPMA. First of all, there is the ontology used to request the mobility services, then an interaction protocol used in Main Migration Protocol (MMP) of IPMA, and, finally, several ontologies used within the migration architecture, and the different migration protocols proposed along the thesis. These ontologies have been kept as simple as possible in order to reduce the data overhead they may represent in an agent migration [Gav04].

A.1 Service registration

The migration architecture, as explained in Section 3.3.2, is registered as a service in the FIPA Directory Facilitator [FIP04], which is implemented by the *df* agent. A `service-description` concept is fulfilled, and sent to this agent. The concept is encapsulated within the `services` field of the `df-agent-description` concept, which, in turn, is put inside a `register` action. All the protocols, languages, and ontologies supported by the service must be listed in the appropriate fields of the `service-description` in addition to the `df-agent-description`.

The architecture is registered as an `amm` agent service. Its name and type are `ipms` and `mobility`. The interaction protocol supported by the service is the `synchronized-request`. The supported ontologies are the `mobility-ontology`, and the `ipms-ontology`. And the content language is the `fipa-sl0`. Furthermore,

other interaction protocols and ontologies must be added depending on the migration protocols used in the Protocol Sequencess (PSs) part of IPMA, e.g. the `fipa-request` interaction protocol, and the `pctp-ontology` for the Push Cache Transfer Protocol (PCTP).

The supported profiles and available migration protocols for the PS must be included in the `service-description` concept. Two properties within the `properties` field are used. The first property is called `agent-profiles`, and includes the set of supported agent profiles by the AM. The `mobile-agent-profile` concept, part of the `ipms-ontology`, is used.

The second property is called `protocols`, and includes the the protocols supported by the middleware in the PS. A specially created `protocols-description` concept of the `mobility-ontology` is used (see Table A.5). This concept contains five fields. The first three fields, which are called `pre-transfer`, `transfer`, and `post-transfer`, are sets of `string` with the migration protocol names. And the other two fields are the `use-preference` and the `relations`, which establish a set of rules and constrains regarding the accepted migration protocols.

- The first field, `relations`, is a set of tuples with protocols that must be executed together, i.e. if one of them is selected, the others present in the tuple should be selected too. It is structured as a set of `term` which contain sets of `string` that compose the mentioned tuples.

- And, finally, the `use-preference` field defines protocol rules to establish protocol priority or obligation of use. The property value is a set of `term` which contains sequences of `string`. Sequences with only one element define the mandatory protocols. In case of sequences with several elements, only one of the protocols of the sequence is mandatory, taking into account that the first is preferred in front of the second, and so on. In case the only aim would be suggesting a preference, the last must be the reserved value `none`. Then, one must choose one of the protocols or none of them.

These two last properties allow the agent middleware to express that some protocols

are preferred in front of others, e.g. authentication method 1 can be preferred to authentication method 2; and that some protocols must be executed together with others, e.g. authentication method 1 must be used together with a resource negotiation.

In the next lines there is an example of a registration message sent from the Agent Mobility Manager (AMM) agent to the DF agent. The message requests the register of the Inter-Platform Mobility Service (IPMS) including PCTP, the On-Demand Transfer Protocol (ODTP), and the One-Shot Agent Authentication Protocol (OSAAP) protocols. In this case it has been specified that PCTP or ODTP must be used, and that, in case of using PCTP it must be used together with OSAAP.

```
(request
  :sender
   (agent-identifier
     :name amm@foo.com
     :addresses (sequence http://foo.com:7778/acc))
  :receiver (set
   (agent-identifier
     :name df@foo.com
     :addresses (sequence http://foo.com:7778/acc)))
  :language fipa-sl0
  :protocol fipa-request
  :ontology fipa-agent-management
  :content
   "((action
     (agent-identifier
       :name df@foo.com
       :addresses (sequence http://foo.com:7778/acc))
     (register
       (df-agent-description
         :name
          (agent-identifier
            :name amm@foo.com
            :addresses (sequence http://foo.com:7778/acc))
         :protocols (set synchronized-request fipa-request)
         :ontologies (set mobility-ontology ipma-ontology pctp-ontology
   )
         :languages (set fipa-sl0)
```

```
:services (set
 (service-description
  :name ipms
  :type mobility
  :protocols (set synchronized-request fipa-request)
  :ontologies (set meeting-scheduler)
  :languages (set fipa-sl0)
  :properties (set
   (property
    :name agent-profiles
    :value (set (mobile-agent-profile (...)))
   (property
    :name protocols
    :value
     (protocols-description
      :pre-transfer (set osaap-v1)
      :transfer (set pctp-v1 odtp-v1)
      :post-transfer
      :use-preference (set (set (osaap-v1 pctp-v1)))
      :relations (set (set (pctp-v1 odtp-v1)))))))))")
```

A.2 Mobility Ontology

The mobility ontology is used in the exchange of messages between ordinary agents and the AMM. Its main purpose is providing the appropriate semantics to regular agents for requesting agent migrations and agent clonning.

Its name is `mobility-ontology`, it is composed of two actions: `move` and `clone` (see Table A.1 and Table A.2); two concepts: `migration-description` and `property` (see Table A.3 and Table A.4 respectively); and one predicate for the failure message (Table A.6). Furthermore, there is the `protocols-description` concept (see Table A.5) used to register the service (see Appendix A.1).

The migration of an agent is requested by sending an ACL request message with the action to perform. The action includes the `migration-description` concept. This concept contains the agent identification, the migration protocols to use, some

specific properties of them (in this case the `property` concept is used), and the number of suggested migration retries AMM should do before considering a migration failed. Furthermore, a remote agent identification can be provided to choose the agent name in case of clonning.

After sending the request, the agent should wait for a response of the migration or cloning success (*inform* or *failure* message). Notices that the failure message contains an exception predicate (see Table A.6). This message is always received from the local AMM regarding the requester agent location (steps 1 and 6 of Figure 3.2 depict these exchanged messages in case of a successful migration). Note that no interaction protocols are used since the only essential message is the first one, and responses are not always received from the same AMM.

A.3 Synchronized Request Interaction Protocol

Sometimes there are processes or actions that must be preceded by other ones. One common case is when two consecutive actions must be done. This is the model required in the IPMA MMP. Since there were no agent interaction protocols which allowed this, a new one has been proposed.

The new interaction protocol is identified by the token `synchronized-request`, which is used in the `protocol` parameter of the ACL message. The interaction protocol, as it is shown in Figure A.1, starts like the IEEE-FIPA Request Interaction Protocol. The main difference is that when the action requested has successfully terminated, another one is immediately requested within the same protocol. In case of the first action failure, the second one is not requested. Notice, that an optional `agree` or `refuse` message can be sent after the first request.

Since the protocol uses two messages with a repeated performative, one of them correspondingly to the first message sent, they must be appropriately distinguished. The user-defined message parameter `X-Action-Order` is used. In the first request message exchanged its value is the string `first`, whereas in the second one it is the string `second`. Finally, all the other rules, such as the conversation-id, cancellation

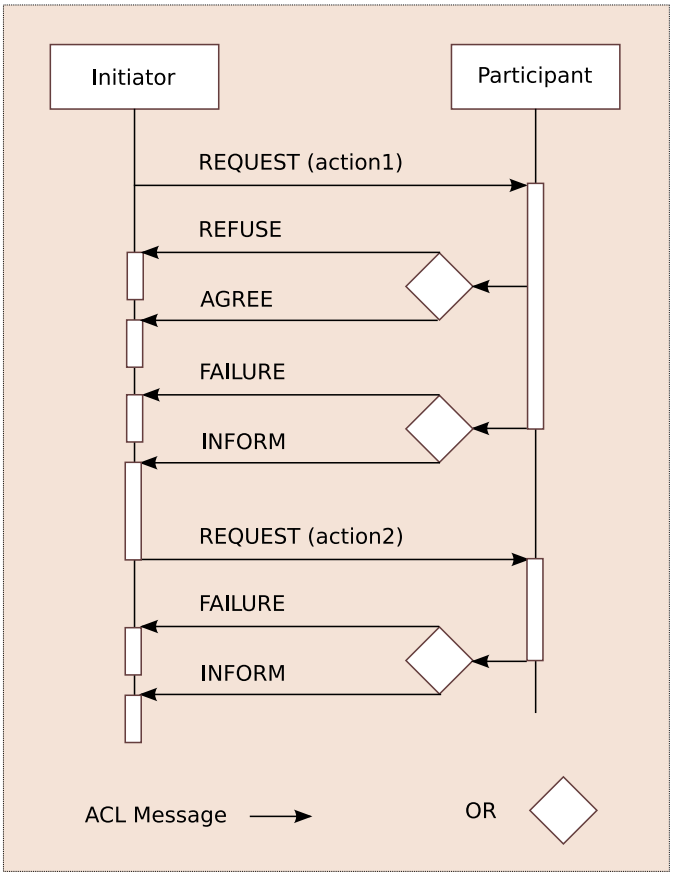

Figure A.1: Synchronized Request Interaction Protocol.

methods, among others, are the same specified for the FIPA Request Interaction Protocol [FIP02k].

A.4 IPMS Ontology

The IPMS ontology is used in the exchange of messages between AMMs of the different locations involved in an agent migration. Its main purpose is to allow AMMs to coordinate the operations of the MMP to perform the agent migration.

Its name is `ipms-ontology`, it is composed of three actions: `move`, `clone` and `resume` (see Table A.7, Table A.8 and Table A.9 respectively); six concepts: `mobile-agent-description`, `mobile-agent-profile`,

`mobile-agent-system`, `mobile-agent-language`, `mobile-agent-os`, and `property` (see Table A.10, Table A.11, Table A.12, Table A.13, Table A.14 and Table A.15 respectively); and two predicates for the refuse and failure messages (see Table A.17 and Table A.16).

The migration process between two AMMs is started by issuing an ACL message which contains the action to perform (`move` or `clone`). The action includes a description of the migrating agent within the `mobile-agent-description` concept. Furthermore this concept includes one or more agent profiles that are composed of all the other concepts mentioned. Once the agent has been transfered and registered into the destination location, a request message with the `resume` action is sent to request its resumption.

A.5 Push Cache Transfer Protocol Ontology

The ontology presented in this section is used in the exchange of messages between AMMs performing agent migrations with PCTP. Its main purpose is the coordination of the protocol and the encapsulation of the agent code, data and state.

Its name is `pctp-ontology`, it is composed of two functions: `transfer-data-state` and `transfer-code` (see Table A.18 and Table A.20 respectively); two predicates to state if the agent code is needed (see Table A.22); and two concepts: `pctp-data-state`, `pctp-code` (see Table A.19, and Table A.21) to encapsulate the agent code, data, and state; and several predicates for the failure message (see Table A.23).

A.6 On Demand Transfer Protocol Ontology

The ontologies presented in this section are used in the exchange of messages between AMMs performing agent migrations with ODTP. Two ontologies are used, one for transferring the agent data and state, and the other to support the on demand code fetching.

The main purpose of the first one is the coordination of the protocol and the encapsulation of the agent data and state. Its name is `odtp-ontology`, it is composed of one function: `transfer-data-state` (see Table A.24); three concepts: `data-state`, `resource-list`, and `resource-description` (see Table A.25, Table A.26, and Table A.27); and several predicates for the failure message (see Table A.28).

The main purpose of the second ontology is the coordination of the agent resource fetching and the encapsulation of each one. Its name is `odtp-fetch-ontology`, it is composed of one function: `fetch-resource` (see Table A.29); one predicate: `resource-fetched` (see Table A.30); one concept: `resource-description` (see Table A.31); and several predicates for the failure message (see Table A.32).

A.7 Fragment Transfer Protocol Ontology

The ontology presented in this section is used in the exchange of messages between AMMs performing agent migrations with the Fragmented Transfer Protocol (FrTP). Its main purpose is the coordination of the protocol and the encapsulation of the agent code, data and state in several fragments.

Its name is `ftp-ontology`, it is composed of two functions: `transfer-agent` and `request-fragment` (see Table A.33 and Table A.38 respectively); two predicates to state if the agent code is needed (see Table A.35); one predicate to encapsulate each fragment, `fragment` (see Table A.37); two concepts, one to negotiate the transference parameters, `parameters` (see Table A.34), and the other to transport and request specific fragments, `fragment-description` (see Table A.39); and several predicates to state the possible errors during the transference (see Tables A.36 and A.40).

A.8 REST Transfer Protocol Ontology

The ontology presented in this section is used in the exchange of messages between AMMs performing agent migrations with the REST Transfer Protocol (RESTTP). Its main purpose is the coordination of the protocol to transfer the agent code, data, and

state using an HTTP request.

Its name is `resttp-ontology`, it is composed of one function, the `transfer-parameters` (see Table A.41); one concept: `rest-parameters` (see Table A.42), and several predicates to represent the protocols errors (see Table A.43).

A.9 One-Shot Agent Authentication Protocol Ontology

The ontology presented in this section is used in the exchange of messages between AMMs performing agent migrations with the One-Shot Agent Authentication Protocol (OSAAP). Its main purpose is the coordination of the protocol, and the encapsulation of the agent owner certificate and agent, and code signatures.

Its name is `osaap-ontology`, it is composed of one function: `x509-agent-auth` (see Table A.44) and two concepts: `x509-agent-auth-description`, and `auth-pair` (see Table A.45, and Table A.46). Furthermore, several predicates are provided to represent the possible authentication negative responses (see Table A.47).

Function	`move`
Ontology	`mobility-ontology`
Supported by	amm
Description	An agent issues a `move` request to start an agent migration process to a remote platform.
Domain	`migration-description`
Arity	1

Table A.1: Move action Mobility Ontology

Function	`clone`
Ontology	`mobility-ontology`
Supported by	amm
Description	An agent issues a `clone` request to start an agent clone process to a remote platform.
Domain	`migration-description`
Arity	1

Table A.2: Clone action Mobility Ontology

Frame	`migration-description`		
Ontology	`mobility-ontology`		
Parameter	**Description**	**Presence**	**Type**
`local-aid`	Name of the agent to migrate.	Mandatory	`agent-identifier`
`remote-aid`	Name assigned to the agent in the remote location.	Optional	`agent-identifier`
`pre-transfer`	Pre-transfer protocols to use.	Optional	Set of `string`
`transfer`	Transfer protocols to use.	Optional	`string`
`post-transfer`	Post-transfer protocols to use.	Optional	Set of `string`
`properties`	Migration properties.	Optional	Set of `property`
`retries`	Suggested migration retries.	Optional	`integer`

Table A.3: Migration Description concept Mobility Ontology

Frame	`property`		
Ontology	`mobility-ontology`		
Parameter	**Description**	**Presence**	**Type**
`name`	Property name.	Mandatory	`string`
`value`	Value.	Mandatory	`term`

Table A.4: Property concept Mobility Ontology

Frame	protocols-description		
Ontology	mobility-ontology		
Parameter	**Description**	**Presence**	**Type**
pre-transfer	Pre-transfer protocols to use.	Optional	Set of string
transfer	Transfer protocols to use.	Optional	string
post-transfer	Post-transfer protocols to use.	Optional	Set of string
relations	Relations between protocols.	Optional	Set of term
use-preference	Execution protocols preferences.	Optional	Set of term

Table A.5: Protocols Description concept Mobility Ontology

Communicative Act	failure	
Ontology	mobility-ontology	
Predicate symbol	**Arguments**	**Description**
agent-already-present	string	Agent is already present in the remote location.
protocols-not-supported	Set of string	Not supported protocols.
protocols-not-allowed	Set of string	Not allowed protocols.
protocols-needed	Set of term	Mandatory protocols.
protocol-combination	Set of term	Protocols on the list are the combinations allowed.
ma-system-not-supported	string	Mobile agent system not supported.
ma-os-not-supported	string	Mobile agent operating system not supported.
ma-language-not-supported	string	Mobile agent language not supported.
protocol-error	string	Error with the indicated protocol.
registration-error	string	Error registering agent to the remote location.
resumption-error	string	Error resuming agent in the remote location.
message-error	string	Semantic error in the message received.
interaction-protocol-error	string	Interaction protocol error.
unknown-error	string	Unknown error.

Table A.6: Failure predicates Mobility Ontology

Function	move
Ontology	ipms-ontology
Supported by	amm
Description	The local AMM issues a move request to a remote AMM to start an agent migration process to a remote platform.
Domain	migration-description
Arity	1

Table A.7: Move action IPMS Ontology

Function	`clone`
Ontology	`ipms-ontology`
Supported by	amm
Description	The AMM issues a `clone` request to a remote AMM to start an agent migration process to a remote platform which ends up with two agents.
Domain	`migration-description`
Arity	1

Table A.8: Clone action IPMS Ontology

Function	`resume`
Ontology	`ipms-ontology`
Supported by	amm
Description	The AMM issues a `resume` request to the remote AMM to resume the agent execution in the remote platform.
Domain	`migration-description`
Arity	1

Table A.9: Resume action IPMS Ontology

Frame	`mobile-agent-description`		
Ontology	`ipms-ontology`		
Parameter	**Description**	**Presence**	**Type**
`mid`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`name`	The unique agent identifier	Mandatory	`agent-identifier`
`agent-profile`	Agent requirements for each provided agent code.	Mandatory	Sequence of `mobile-agent-profile`
`cgid`	Agent code group identification.	Mandatory	`string`
`data-encoding`	Agent data encoding mechanism.	Mandatory	`string`
`state-encoding`	Agent state encoding mechanism.	Optional	`string`
`agent-version`	Agent version.	Optional	`string`
`pre-transfer`	Pre-transfer protocols chosen	Optional	Sequence of `string`
`transfer`	Transfer protocols chosen	Mandatory	`string`
`post-transfer`	Post-transfer protocols chosen	Optional	Sequence of `string`

Table A.10: Mobile Agent Description concept IPMS Ontology

Frame	`mobile-agent-profile`		
Ontology	`ipms-ontology`		
Parameter	**Description**	**Presence**	**Type**
`cid`	Agent Code Identification.	Mandatory	`string`
`system`	Mobile agent system environment.	Mandatory	`mobile-agent-system`
`language`	Language environment.	Optional	`mobile-agent-language`
`os`	Operating system environment.	Optional	`mobile-agent-os`

Table A.11: Mobile Agent Profile concept IPMS Ontology

Frame	`mobile-agent-system`		
Ontology	`ipms-ontology`		
Parameter	**Description**	**Presence**	**Type**
`name`	Mobile agent system name.	Mandatory	`string`
`major-version`	Major version.	Mandatory	`string`
`minor-version`	Minor version.	Optional	`string`
`dependencies`	Dependencies required.	Optional	Set of `property`

Table A.12: Mobile Agent System concept IPMS Ontology

Frame	`mobile-agent-language`		
Ontology	`ipms-ontology`		
Parameter	**Description**	**Presence**	**Type**
`name`	Mobile agent PL name.	Mandatory	`string`
`major-version`	Major version.	Mandatory	`string`
`minor-version`	Minor version.	Optional	`string`
`format`	Code base format.	Optional	`string`
`filter`	Filter to execute over the code base before execute.	Optional	`string`
`dependencies`	Language dependencies.	Optional	Set of `property`

Table A.13: Mobile Agent Language concept IPMS Ontology

Frame	`mobile-agent-os`		
Ontology	`ipms-ontology`		
Parameter	**Description**	**Presence**	**Type**
`name`	Operating system name.	Mandatory	`string`
`major-version`	Major version.	Mandatory	`string`
`minor-version`	Minor version.	Optional	`string`
`hardware`	Hardware below operating system.	Optional	`string`
`dependencies`	Dependencies required.	Optional	Set of `property`

Table A.14: Mobile Agent OS concept IPMS Ontology

Frame	property		
Ontology	ipms-ontology		
Parameter	**Description**	**Presence**	**Type**
name	Property name.	Mandatory	string
value	Value.	Mandatory	term

Table A.15: Property concept IPMS Ontology

Communicative Act	refuse	
Ontology	ipms-ontology	
Predicate symbol	**Arguments**	**Description**
in-use-mid	string	The selected MID it is being used in another migration transaction.
agent-already-present	string	Agent is already present in the remote location.
protocols-not-supported	Set of string	Protocols on the list are not supported.
protocols-not-allowed	Set of string	Protocols on the list are not allowed.
protocols-needed	Set of term	Protocols on the list are mandatory.
protocol-combination	Set of term	Protocols combination not supported. Protocols on the list are the combinations allowed.
ma-system-not-supported	string	Mobile agent system is not supported.
ma-os-not-supported	string	Mobile agent operating system is not supported.
ma-language-not-supported	string	Mobile agent language is not supported.

Table A.16: Refuse predicates IPMS Ontology

Communicative Act	failure	
Ontology	ipms-ontology	
Predicate symbol	**Arguments**	**Description**
protocol-error	string	Protocol error in the indicated one.
registration-error	string	Error registering agent to the remote location.
resumption-error	string	Error resuming agent execution in the remote location.
message-error	string	Semantic error in the message received.
interaction-protocol-error	string	Interaction protocol error.
unknown-error	string	Unknown error.

Table A.17: Failure predicates IPMS Ontology

Function	`transfer-data-state`
Ontology	`pctp-ontology`
Supported by	amm
Description	The AMM issues a `transfer-data-state` request to transfer the agent data and state to the remote platform.
Domain	`data-state`
Arity	1

Table A.18: Transfer Data State action PCTP Ontology

Frame	`data-state`		
Ontology	`pctp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`mid`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`cid`	Agent code identification.	Mandatory	`string`
`sr`	Agent code security revision.	Mandatory	`string`
`hcid`	Agent code hash.	Mandatory	`string`
`data`	Agent data.	Mandatory	`byte-stream`
`state`	Agent state.	Optional	`byte-stream`

Table A.19: Data State concept PCTP Ontology

Function	`transfer-code`
Ontology	`pctp-ontology`
Supported by	amm
Description	The AMM issues a `transfer-code` request to to transfer the agent code to the remote platform.
Domain	`code`
Arity	1

Table A.20: Transfer Code action PCTP Ontology

Frame	`code`		
Ontology	`pctp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`mid`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`cid`	Agent code identification.	Mandatory	`string`
`sr`	Agent code security revision.	Mandatory	`string`
`hcid`	Agent code hash.	Mandatory	`string`
`code`	Agent code.	Mandatory	`byte-stream`

Table A.21: Code concept PCTP Ontology

Communicative Act	`inform`	
Ontology	`pctp-ontology`	
Predicate symbol	**Arguments**	**Description**
`code-is-needed`		The agent code is not present in the remote platform and it must be sent.
`code-is-not-needed`		The agent code is already present in the remote platform and it does not have to be sent.

Table A.22: Inform predicates PCTP Ontology

Communicative Act	`failure`	
Ontology	`pctp-ontology`	
Predicate symbol	**Arguments**	**Description**
`invalid-mid`	`string`	The Migration Identifier (MID) is not valid.
`not-enough-space`	`string`	There is not enough space in the remote location.
`data-error`	`string`	Error in the provided data.
`state-error`	`string`	Error in the provided state.
`code-error`	`string`	Error in the provided code.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.23: Failure predicates PCTP Ontology

Function	`transfer-data-state`
Ontology	`odtp-ontology`
Supported by	amm
Description	The AMM issues a `transfer-data-state` request to transfer the agent data and state to the remote platform.
Domain	`data-state`
Arity	1

Table A.24: Transfer Data State action ODTP Ontology

Frame	`data-state`		
Ontology	`odtp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`mid`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`cid`	Agent code identification.	Mandatory	`string`
`sr`	Agent code security revision.	Mandatory	`string`
`hcid`	Agent code hash.	Mandatory	`string`
`data`	Agent data.	Mandatory	`byte-stream`
`state`	Agent state.	Optional	`byte-stream`
`resource-list`	List of agent resources.	Mandatory	`resource-list`
`resource-servers`	List of resource servers.	Mandatory	Sequence of `string`

Table A.25: Data State concept ODTP Ontology

Frame	`resource-list`		
Ontology	`odtp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`resources`	List of agent resources.	Mandatory	Sequence of `resource-description`
`hash-algorithm`	Hash algorithm used in each resource.	Mandatory	`string`

Table A.26: Resource List concept ODTP Ontology

Frame	`resource-description`		
Ontology	`odtp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`name`	Resource name.	Mandatory	`string`
`hash`	Resource hash.	Mandatory	`string`

Table A.27: Resource Description concept ODTP Ontology

Communicative Act	`failure`	
Ontology	`odtp-ontology`	
Predicate symbol	**Arguments**	**Description**
`invalid-mid`	`string`	The MID is not valid.
`incorrect-resource-list`	`string`	Incorrect resource list.
`incorrect-resource-server`	`string`	Incorrect resource server.
`not-enough-space`	`string`	There is not enough space in the remote location.
`data-error`	`string`	Error with the data provided.
`state-error`	`string`	Error with the state provided.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.28: Failure predicates ODTP Ontology

Function	`fetch-resource`
Ontology	`odtp-fetch-ontology`
Supported by	amm
Description	The AMM issues a `fetch-resource` request to ask for an agent resource in the agent resources middleware.
Domain	`resource-description`
Arity	1

Table A.29: Fetch Resource action ODTP Fetch Ontology

Communicative Act	`inform`	
Ontology	`odtp-fetch-ontology`	
Predicate symbol	**Arguments**	**Description**
`resource-fetched`	`byte-stream`	Contains the requested result.

Table A.30: Inform predicates ODTP Fetch Ontology

Frame	`resource-description`		
Ontology	`odtp-fetch-ontology`		
Parameter	**Description**	**Presence**	**Type**
`hash`	Resource hash.	Mandatory	`string`
`algorithm`	Algorithm used for the resource hash.	Mandatory	`string`

Table A.31: Resource Description concept ODTP Fetch Ontology

Communicative Act	`failure`	
Ontology	`odtp-fetch-ontology`	
Predicate symbol	**Arguments**	**Description**
`resource-not-found`	`string`	Requested resource not found.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.32: Failure predicates ODTP Fetch ontology

Function	`transfer-agent`
Ontology	`ftp-ontology`
Supported by	amm
Description	The AMM issues a `request-transfer-agent` request to transfer the agent to the remote platform
Domain	`parameters`
Arity	1

Table A.33: Request Transfer Agent action FrTP Ontology

Frame	`parameters`		
Ontology	`ftp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`fragment-size`	The fragment size which the origin platform wants to apply in the migration process	Mandatory	`integer`
`code-size`	The agent code size	Mandatory	`integer`
`data-size`	Agent instance size	Mandatory	`integer`
`state`	Agent state	Optional	`byte-stream`
`cid`	Agent code identification.	Mandatory	`string`
`sr`	Agent code security revision.	Mandatory	`string`
`hcid`	Agent code hash.	Mandatory	`string`

Table A.34: Parameters concept FrTP Ontology

Communicative Act	`agree`	
Ontology	`ftp-ontology`	
Predicate symbol	**Arguments**	**Description**
`code-is-needed`		The agent code is not present in the remote platform and it must be sent.
`code-is-not-needed`		The agent code is already present in the remote platform and it does not have to be sent.

Table A.35: Agree predicates FrTP Ontology

Communicative Act	`refuse`	
Ontology	`ftp-ontology`	
Predicate symbol	**Arguments**	**Description**
`not-enough-space`	`string`	There is not enough space in the remote platform.
`agent-too-big`	`string`	Agent size is too big and can not be accepted in the remote platform.
`reject-fragment-size`	`string`	Fragment size is not suitable.

Table A.36: Refuse predicates FrTP Ontology

Communicative Act	`inform`	
Ontology	`ftp-ontology`	
Predicate symbol	**Arguments**	**Description**
`fragment`	`fragment-description`	Message filled with agent fragment.

Table A.37: Inform predicates FrTP Ontology

Function	`request-fragment`
Ontology	`ftp-ontology`
Supported by	amm
Description	The AMM issues a `request-fragment` request to transfer a lost fragment to the remote platform
Domain	`fragment-description`
Arity	1

Table A.38: Request Fragment action FrTP Ontology

Frame	`fragment-description`		
Ontology	`ftp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`content`	Agent code or data snippet which fits to the fragment.	Optional	`byte-stream`
`id`	Fragment number that identifies it in the migration.	Mandatory	`integer`
`type`	Indicates the type of fragment: `data` or `code`	Mandatory	`string`
`mid`	Migration transaction in which the fragment is associated.	Mandatory	`string`

Table A.39: Fragment Description concept FrTP Ontology

Communicative Act	`failure`	
Ontology	`ftp-ontology`	
Predicate symbol	**Arguments**	**Description**
`instance-size-error`	`string`	Error instance size not found.
`code-size-error`	`string`	Error code size not found.
`fragment-size-error`	`string`	Error fragment size not found.
`cid-error`	`string`	Error cid not found.
`bad-formed-msg-error`	`string`	Message received not properly created.
`extracting-content-error`	`string`	Error extracting message content.
`registration-error`	`string`	Error registering agent to the remote location.
`action-error`	`string`	Action received is not valid.
`null-action-error`	`string`	Null action received.
`protocol-error`	`string`	Migration protocol is not valid.
`agent-entry-error`	`string`	Agent entry not found to the remote location.
`migration-service-error`	`string`	Error contacting migration service to notice failure.
`out-of-sequence-error`	`string`	Message received is out of sequence.
`fragment-id-error`	`string`	Not valid fragment number identifier.
`fragment-aid-error`	`string`	Not valid agent unique identifier.
`fragment-type-error`	`string`	Not valid fragment type.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.40: Failure predicates FrTP Ontology

Function	`transfer-parameters`
Ontology	`resttp-ontology`
Supported by	amm
Description	The AMM agent issues a `transfer-parameters` request to transfer the parameters required by the remote location to fetch the agent code, data and state.
Domain	`rest-parameters`
Arity	1

Table A.41: Transfer Parameters action RESTTP Ontology

Frame	`rest-parameters`		
Ontology	`resttp-ontology`		
Parameter	**Description**	**Presence**	**Type**
`mid`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`host`	HTTP server hostname.	Mandatory	`string`
`port`	HTTP server port.	Mandatory	`string`
`ssl-enabled`	Enables the use of SSL in the HTTP request.	Optional	`boolean`
`code-nonce`	Agent code identification.	Mandatory	`string`
`data-nonce`	Agent data identification.	Mandatory	`string`
`state-nonce`	Agent state identification.	Optional	`string`
`cid`	Agent code identification.	Mandatory	`string`
`sr`	Agent code security revision.	Mandatory	`string`
`hcid`	Agent code hash.	Mandatory	`string`

Table A.42: Rest Parameters concept RESTTP Ontology

Communicative Act	`failure`	
Ontology	`resttp-ontology`	
Predicate symbol	**Arguments**	**Description**
`invalid-migration-id`	`string`	Migration ID is not valid.
`not-enough-space`	`string`	There is not enough space in the remote location.
`data-error`	`string`	Error with obtained data.
`state-error`	`string`	Error with obtained state.
`code-error`	`string`	Error with obtained code.
`data-not-available`	`string`	Error getting data.
`state-not-available`	`string`	Error getting state.
`code-not-available`	`string`	Error getting code.
`ssl-error`	`string`	SSL not supported by the HTTP server.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.43: Failure predicates RESTTP Ontology

Function	`x509-agent-auth`
Ontology	`osaap-ontology`
Supported by	amm
Description	The AMM agent issues a `x509-agent-auth` request to authenticate the agent in the remote platform.
Domain	`x509-agent-auth-description`
Arity	1

Table A.44: X509 Agent Auth action OSAAP Ontology

Frame	`x509-agent-auth-description`		
Ontology	`osaap-ontology`		
Parameter	**Description**	**Presence**	**Type**
`migration-id`	Unique migration identifier between the two involved agent middleware	Mandatory	`string`
`agent-signature`	Agent owner certificate and agent signature encapsulated in a PKCS7 data structure	Mandatory	`byte-stream`
`code-signatures`	Set of code signatures encapsulated in a PKCS7 data structures.	Optional	Set of `auth-pair`

Table A.45: X509 Agent Auth Description concept OSAAP Ontology

Frame	`auth-pair`		
Ontology	`osaap-ontology`		
Parameter	**Description**	**Presence**	**Type**
`key`	Identifier of the signed entity.	Mandatory	`string`
`signature`	Signature encapsulated in a PKCS7 data structure.	Mandatory	`byte-stream`

Table A.46: Auth Pair concept OSAAP Ontology

Communicative Act	`failure`	
Ontology	`osaap-ontology`	
Predicate symbol	**Arguments**	**Description**
`invalid-mid`	`string`	Migration ID is not valid.
`certificate-not-found`	`string`	The agent owner certificate is not included in the message and cannot be found anywhere.
`corrupted-certificate`	`string`	The agent certificate is corrupted.
`invalid-agent-signature`	`string`	The agent signature does not validate.
`corrupted-agent-signature`	`string`	The agent signature is corrupted.
`corrupted-code-signatures`	`string`	The code signatures are corrupted.
`message-error`	`string`	Semantic error in the message received.
`interaction-protocol-error`	`string`	Interaction protocol error.
`unknown-error`	`string`	Unknown error.

Table A.47: Failure predicates OSAAP Ontology

Appendix B

Common Agent Interface

This appendix contains two specific realisations of the Common Agent Interface (CAI) for the Java and Python Programming Languages (PLs). The two of them can be used to develop equivalently structured agent codes that can help in challenge of providing full agent interoperability.

B.1 Java CAI

```
1  package fipa.api;
2
3  public abstract class Agent implements Serializable {
4
5
6        // METHODS IMPLEMENTED BY DEFAULT
7        public Agent(AID aid) {
8              this.aid = aid;
9        }
10
11       public abstract void deliver(ACL m);
12
13       public void setAgentState(int s) {
14             state = s;
15       }
```

```
16
17        public int getAgentState() {
18              return state;
19        }
20
21        public void setAID(AID a) {
22              aid = a;
23        }
24
25        public AID getAID() {
26              return aid;
27        }
28
29        public abstract void run();
30
31        public void setMContext(MiddlewareContext ac) {
32              aContext = ac;
33        }
34
35        public MiddlewareContext getMContext() {
36              return aContext;
37        }
38
39        // API ATTRIBUTES (Subject to serialization)
40        private int state;
41
42        private AID aid;
43
44        // NON-API ATTRIBUTES
45        private transient MiddlewareContext aContext;
46
47  }
48
49
50  public interface MiddlewareContext {
51
52        public void send(ACL m);
53
```

```
54        public AID getAMS();
55
56  }
57
58
59  public class AID {
60
61        public AID() {
62
63        }
64
65        public AID(String name) {
66              this.name = name;
67              addresses = new Vector<String>();
68              resolvers = new Vector<AID>();
69        }
70
71        public String getName() {
72              return name;
73        }
74
75        public void addAddress (String a) {
76              addresses.add(a);
77        }
78
79        public Iterator<String> getAllAddresses() {
80              return addresses.iterator();
81        }
82
83        public boolean removeAddress(String a) {
84              return addresses.remove(a);
85        }
86
87        public void clearAddresses() {
88              addresses.clear();
89        }
90
91        public void addResolver(AID r) {
```

```
92              resolvers.add(r);
93        }
94
95        public Iterator<AID> getAllResolvers() {
96              return resolvers.iterator();
97        }
98
99        public boolean removeResolver(AID r) {
100             return resolvers.remove(r);
101       }
102
103       public void clearResolvers() {
104             resolvers.clear();
105       }
106
107
108
109       public String name;
110
111       public Vector<String> addresses;
112
113       public Vector<AID> resolvers;
114
115 }
116
117
118 public class ACL {
119
120
121       // FIPA PERFORMATIVES
122       public static final String ACCEPT_PROPOSAL = "accept-proposal";
123       public static final String AGREE = "agree";
124       public static final String CANCEL = "cancel";
125       public static final String CALL_FOR_PROPOSAL = "cfp";
126       public static final String CONFIRM = "confirm";
127       public static final String DISCONFIRM = "disconfirm";
128       public static final String FAILURE = "failure";
129       public static final String INFORM = "inform";
```

```
130         public static final String INFORM_IF = "inform-if";
131         public static final String INFORM_REF = "inform-ref";
132         public static final String NOT_UNDERSTOOD = "not-understood";
133         public static final String PROPAGATE = "propagate";
134         public static final String PROPOSE = "propose";
135         public static final String PROXY = "proxy";
136         public static final String QUERY_IF = "query-if";
137         public static final String QUERY_REF = "query-ref";
138         public static final String REFUSE = "refuse";
139         public static final String REJECT_PROPOSAL = "reject-proposal";
140         public static final String REQUEST = "request";
141         public static final String REQUEST_WHEN = "request-when";
142         public static final String REQUEST_WHENEVER = "request-whenever
              ";
143         public static final String SUBSCRIBE = "subscribe";
144
145
146         public ACL() {
147               receivers = new Vector<AID>();
148               ud_parameters = new Hashtable<String,String>();
149         }
150
151         // API METHODS
152         public void setPerformative(String p) {
153               performative = p;
154         }
155
156         public String getPerformative() {
157               return performative;
158         }
159
160         public void setSender(AID aid) {
161               sender = aid;
162         }
163
164         public AID getSender() {
165               return sender;
166         }
```

```
167
168       public void addReceiver(AID r) {
169             receivers.add(r);
170       }
171
172       public Iterator<AID> getAllReceivers() {
173             return receivers.iterator();
174       }
175
176       public boolean removeReceiver(AID r) {
177             return receivers.remove(r);
178       }
179
180       public void clearReceivers() {
181             receivers.clear();
182       }
183
184       public void setReplyTo(AID aid) {
185             replyto = aid;
186       }
187
188       public AID getReplyTo() {
189             return replyto;
190       }
191
192       public void setStringContent(String c) {
193             string_content = c;
194             binary_content = null;
195       }
196
197       public String getStringContent() {
198             return string_content;
199       }
200
201       public void setBinaryContent(byte[] c) {
202             binary_content = c;
203             string_content = null;
204       }
```

```
205
206         public byte[] getBinaryContent() {
207               return binary_content;
208         }
209
210         public void setLanguage(String l) {
211               language = l;
212         }
213
214         public String getLanguage() {
215               return language;
216         }
217
218         public void setEncoding(String e) {
219               encoding = e;
220         }
221
222         public String getEncoding() {
223               return encoding;
224         }
225
226         public void setOntology(String o) {
227               ontology = o;
228         }
229
230         public String getOntology() {
231               return ontology;
232         }
233
234         public void setProtocol(String p) {
235               protocol = p;
236         }
237
238         public String getProtocol() {
239               return protocol;
240         }
241
242         public void setConversationId(String cid) {
```

```
243             conversation_id = cid;
244         }
245
246         public String getConversationId() {
247             return conversation_id;
248         }
249
250         public void setReplyWith(String rw) {
251             reply_with = rw;
252         }
253
254         public String getReplyWith() {
255             return reply_with;
256         }
257
258         public void setInReplyTo(String irt) {
259             in_reply_to = irt;
260         }
261
262         public String getInReplyTo() {
263             return in_reply_to;
264         }
265
266         public void setReplyBy(String rb) {
267             reply_by = rb;
268         }
269
270         public String getReplyBy() {
271             return reply_by;
272         }
273
274         public void addUserDefinedParameter(String name, String value)
                {
275             ud_parameters.put(name, value);
276         }
277
278         public String getUserDefinedParameter(String name) {
279             return ud_parameters.get(name);
```

```
280         }
281
282         public String removeUserDefinedParameter(String name) {
283               return ud_parameters.remove(name);
284         }
285
286         public void setACLEncoding(String e) {
287               acl_encoding = e;
288         }
289
290         public String getACLEncoding() {
291               return acl_encoding;
292         }
293
294         public ACL createReply() {
295
296               ACL acl = new ACL();
297
298               acl.acl_encoding = this.acl_encoding;
299               acl.binary_content = this.binary_content;
300               acl.string_content = this.string_content;
301               acl.conversation_id = this.conversation_id;
302               acl.encoding = this.encoding;
303               acl.in_reply_to = this.reply_with;
304               acl.language = this.language;
305               acl.ontology = this.ontology;
306               acl.receivers.add(this.sender);
307
308               return acl;
309         }
310
311         // API ATTRIBUTES
312         public String performative;
313         public AID sender;
314         public Vector<AID> receivers;
315         public AID replyto;
316         public byte[] binary_content;
317         public String string_content;
```

```
318         public String language;
319         public String encoding;
320         public String ontology;
321         public String protocol;
322         public String conversation_id;
323         public String reply_with;
324         public String in_reply_to;
325         public String reply_by;
326         public Hashtable<String,String> ud_parameters;
327         public String acl_encoding;
328 
329 
330 }
```

B.2 Python CAI

```
1  
2  import yaml
3  
4  
5  class Agent:
6  
7     "FIPA agent base class"
8     def __init__(self, aid): abstract()
9  
10 
11 
12    """ API methods """
13    def run(self): abstract()
14 
15    def deliver(self, msg): abstract()
16 
17    def setMContext(self, ac): abstract()
18 
19    def getMContext(self): abstract()
20 
```

```
21    def setAgentState(self, state): abstract()
22
23    def getAgentState(self): abstract()
24
25    def getAID(self): abstract()
26
27    """ Non-API methods """
28    def __abstract():
29       import inspect
30       caller = inspect.getouterframes(inspect.currentframe())[1][3]
31       raise NotImplementedError(caller + ' must be implemented in
             subclass')
32
33
34 class MiddlewareContext:
35
36    """def blockingReceive(self,timeout,template): abstract()
37
38    def receive(template): abstract()
39
40    def doSuspend(): abstract()
41
42    def doWait(): abstract()"""
43
44    def send(message): abstract()
45
46    def getAMS(): abstract()
47
48
49    """ Non-API methods """
50    def __abstract():
51       import inspect
52       caller = inspect.getouterframes(inspect.currentframe())[1][3]
53       raise NotImplementedError(caller + ' must be implemented in
             subclass')
54
55
56 class AID(yaml.YAMLObject):
```

```
57
58    yaml_tag = u'!AID'
59    def __init__(self, agent_name):
60       self.name = agent_name
61       self.addresses = []
62       self.resolvers = []
63
64    def getName(self):
65       return self.name
66
67    def addAddress(self, address):
68       self.addresses.append(address)
69
70    def getAllAddresses(self):
71       return self.addresses
72
73    def removeAddress(self, address):
74       return (self.addresses.pop(address)==address)
75
76    def clearAddresses(self):
77       self.addresses = []
78
79    def addResolver(self, resolver):
80       self.resolvers.add(resolver);
81
82    def getAllResolvers(self):
83       self.resolvers
84
85    def removeResolver(self, resolver):
86       return (self.resolvers.remove(resolver)==resolver)
87
88    def clearResolvers():
89       self.resolvers = []
90
91
92 class ACL(yaml.YAMLObject):
93
94
```

```
95     """ FIPA PERFORMATIVES """
96     ACCEPT_PROPOSAL = "accept-proposal";
97     AGREE = "agree";
98     CANCEL = "cancel";
99     CALL_FOR_PROPOSAL = "cfp";
100    CONFIRM = "confirm";
101    DISCONFIRM = "disconfirm";
102    FAILURE = "failure";
103    INFORM = "inform";
104    INFORM_IF = "inform-if";
105    INFORM_REF = "inform-ref";
106    NOT_UNDERSTOOD = "not-understood";
107    PROPAGATE = "propagate";
108    PROPOSE = "propose";
109    PROXY = "proxy";
110    QUERY_IF = "query-if";
111    QUERY_REF = "query-ref";
112    REFUSE = "refuse";
113    REJECT_PROPOSAL = "reject-proposal";
114    REQUEST = "request";
115    REQUEST_WHEN = "request-when";
116    REQUEST_WHENEVER = "request-whenever";
117    SUBSCRIBE = "subscribe";
118
119    yaml_tag = u'!ACL'
120    def __init__(self):
121
122       """ API ATTRIBUTES """
123       self.receivers = []
124       self.ud_parameters = {}
125       self.performative = ""
126       self.sender = []
127       self.replyto = ""
128       self.binary_content = ""
129       self.string_content = ""
130       self.language = ""
131       self.encoding = ""
132       self.ontology = ""
```

```
133         self.protocol = ""
134         self.conversation_id = ""
135         self.reply_with = ""
136         self.in_reply_to = ""
137         self.reply_by = ""
138         self.acl_encoding = ""
139 
140 
141     """ API METHODS """
142     def setPerformative(self, perf):
143         self.performative = perf
144 
145     def getPerformative(self):
146         return self.performative
147 
148     def setSender(self, aid):
149         self.sender = aid
150 
151     def getSender(self):
152         self.sender
153 
154     def addReceiver(self, receiver):
155         self.receivers.append(receiver)
156 
157     def getAllReceivers(self):
158         return self.receivers
159 
160     def removeReceiver(self, receiver):
161         return receivers.pop(r);
162 
163     def clearReceivers(self):
164         self.receivers = []
165 
166     def setReplyTo(self, aid):
167         self.replyto = aid;
168 
169     def getReplyTo(self):
170         return self.replyto;
```

```
171
172     def setStringContent(self, content):
173        self.string_content = content
174        self.binary_content = None
175
176     def getStringContent(self):
177        return self.string_content
178
179     def setBinaryContent(self, content):
180        self.binary_content = content
181        self.string_content = None
182
183     def getBinaryContent(self):
184        return self.binary_content
185
186     def setLanguage(self, language):
187        self.language = language
188
189     def getLanguage(self):
190        return self.language
191
192     def setEncoding(self, encoding):
193        self.encoding = encoding
194
195     def getEncoding(self):
196        return self.encoding
197
198     def setOntology(self, ontology):
199        self.ontology = ontology
200
201     def getOntology(self):
202        return self.ontology
203
204     def setProtocol(self, protocol):
205        self.protocol = protocol
206
207     def getProtocol(self):
208        return self.protocol
```

```
209
210     def setConversationId(self, cid):
211        self.conversation_id = cid
212
213     def getConversationId(self):
214        return self.conversation_id;
215
216     def setReplyWith(self, reply_with):
217        self.reply_with = reply_with
218
219     def getReplyWith():
220        return self.reply_with
221
222     def setInReplyTo(self, in_reply_to):
223        self.in_reply_to = irt
224
225     def getInReplyTo(self):
226        return self.in_reply_to
227
228     def setReplyBy(self, reply_by):
229        self.reply_by = reply_by
230
231     def getReplyBy(self):
232        return self.reply_by
233
234     def addUserDefinedParameter(self, name, value):
235        self.ud_parameters.put[name] = value
236
237     def getUserDefinedParameter(self, name):
238        return self.ud_parameters[name]
239
240     def removeUserDefinedParameter(self, name):
241        return self.ud_parameters.pop(name)
242
243     def setACLEncoding(self, encoding):
244        self.acl_encoding = encoding;
245
246     def getACLEncoding(self):
```

```
247         return self.acl_encoding
248 
249     def createReply():
250 
251         acl = ACL()
252 
253         acl.acl_encoding = self.acl_encoding;
254         acl.binary_content = self.binary_content;
255         acl.string_content = self.string_content;
256         acl.conversation_id = self.conversation_id;
257         acl.encoding = self.encoding;
258         acl.in_reply_to = self.reply_with;
259         acl.language = self.language;
260         acl.ontology = self.ontology;
261         acl.receivers.add(self.sender);
262 
263         return acl;
```

Jordi Cucurull Juan

Bellaterra, September 2008