Dottorato di Ricerca in Informatica

Università di Bologna, Padova

A Communication Infrastructure to Support Knowledge Level Agents on the Web

Davide Guidi

March 2007

Coordinatore:	Tutore:
Prof. Özalp Babaoğlu	Prof. Mauro Gaspari

Abstract

Agent Communication Languages (ACLs) have been developed to provide a way for agents to communicate with each other supporting cooperation in Multi-Agent Systems. In the past few years many ACLs have been proposed for Multi-Agent Systems, such as KQML and FIPA-ACL. The goal of these languages is to support high-level, human like communication among agents, exploiting Knowledge Level features rather than symbol level ones. Adopting these ACLs, and mainly the FIPA-ACL specifications, many agent platforms and prototypes have been developed.

Despite these efforts, an important issue in the research on ACLs is still open and concerns how these languages should deal (at the Knowledge Level) with possible failures of agents. Indeed, the notion of Knowledge Level cannot be straightforwardly extended to a distributed framework such as MASs, because problems concerning communication and concurrency may arise when several Knowledge Level agents interact (for example deadlock or starvation).

The main contribution of this Thesis is the design and the implementation of NOWHERE, a platform to support Knowledge Level Agents on the Web. NOWHERE exploits an advanced Agent Communication Language, $\mathbb{FT}$-$\mathbb{ACL}$, which provides high-level fault-tolerant communication primitives and satisfies a set of well defined Knowledge Level programming requirements. NOWHERE is well integrated with current technologies, for example providing full integration for Web services. Supporting different middleware used to send messages, it can be adapted to various scenarios. In this Thesis we present the design and the implementation of the architecture, together with a discussion of the most interesting details and a comparison with other emerging agent platforms. We also present several case studies where we discuss the benefits of programming agents using the NOWHERE architecture, comparing the results with other solutions. Finally, the complete source code of the basic examples can be found in appendix.

Contents

List of Tables

List of Figures

THESIS OUTLINE

After the first Chapter, that provides an introduction, this Thesis is composed of three Parts. In the following we give a brief overview of each Part.

Part I - State of the Art

In this Part we give an overview of the state of the art in Agent Communication Languages and in Multi-Agent Systems.

Chapter 2 first defines some basic concepts used in the Thesis and then provides a snapshot of the current state of the art, presenting the scenario in which we locate the work done in the context of this Thesis.

Part II - The NOWHERE architecture

In this Part we describe in detail the design and the implementation of the NOWHERE architecture.

Chapter 3 introduces the architecture, providing a brief description of its key features.

Chapter 4 presents $\mathbb{FT}$-$\mathbb{ACL}$, the Agent Communication Language used by NOWHERE, providing a detailed description of the core language primitives.

Chapter 5 describes the basic components of an agent: the Dispatcher and the Facilitator.

Chapter 6 analysis some interesting NOWHERE details, comparing them with other agent architectures, such as JADE, when possible. In particular, the following features are highlighted:

- the agent naming mechanism;
- the timeout handling mechanism;

- how NOWHERE can be adapted to different scenarios;
- how NOWHERE manages groups of agents;
- the Web service integration;

Part III - Case studies

In this Part we present three case studies. The first one, Chapter 7.1, provides a detailed comparison of the solutions obtained with the NOWHERE architecture and with a state of the art agent platform regarding the classic Contract Net protocol.

Chapter 7.2 introduces a "gridified" version of the Connect 4 game, showing how this problem can be solved using NOWHERE and then comparing this approach with the IBM Globus Grid toolkit. Furthermore, Chapter 7.3 analyzes the realization of a Distributed Grid Performance System using NOWHERE. An Image Rendering Architecture built on top of the Grid Performance System is then presented, together with some collected results.

Finally, Chapter 8 presents the conclusions and highlights some future work. The complete source code of some basic example can be found in appendix.

THESIS CONTRIBUTION

The main contribution of this Thesis is the design and the implementation of NOWHERE, an architecture that supports Knowledge Level agents on the Web. In particular, the results of this Thesis can be summarized as follows.

- Implementation of $\mathbb{FT}$-$\mathbb{ACL}$, an advanced Agent Communication Language which provides fault tolerant communication primitives maintaining a Knowledge Level characterization of the ACL.
- Design and implementation of NOWHERE, an agent architecture for supporting Knowledge Level agents which uses $\mathbb{FT}$-$\mathbb{ACL}$ as a communication language. The main features of $\mathbb{FT}$-$\mathbb{ACL}$ are:
  - *An open architecture*, which allows the programmer to dynamically integrate new agents into the existing MAS.
  - *A platform for Knowledge Level agents*, where the programmer does not have to handle explicitly many low level issues, such as network and concurrency problems.
  - *Support for interoperability*. NOWHERE agents can be realised in any programming language including AI languages or knowledge representation languages, provided that they react to a well defined protocol based on the standard primitives of $\mathbb{FT}$-$\mathbb{ACL}$
  - *Integration with Web services*. NOWHERE supports a complete Web service integration. Agents have the ability to export capabilities as Web services and they are also able to invoke Web services using the standard ACL primitives. An agent can invoke a service provided by another agent or a Web service in the same way.

  - *Infrastructure adaptable to different scenarios*. NOWHERE is built using three different layers. The network layer used to send messages can be changed without affecting the other parts, so that the platform can be adapted to different scenarios, ranging from a small set of agents that need real time communication, to a huge set of agents over a network with high latency.

- Comparing to other agent platforms, NOWHERE presents a number of innovative aspects, such as:

  - the realization of a transparent timeout mechanism;
  - the internal use of groups of agents interested in specific services, in order to limit the broadcast scope;
  - the integration of Web services.

The material presented in this Thesis has been published in good part in the following papers:

1. *A Fault Tolerant Agent Communication Language for Supporting Web Agent Interaction*
   N. Dragoni, M. Gaspari, D. Guidi, Agent Communication: International Workshop on Agent Communication (AC2005), Revised Selected and Invited Papers, volume 3859, LNAI, Springer Verlag, 2006.

2. *An Infrastructure to Support Cooperation of Knowledge-Level Agents on the Semantic Grid*
   N. Dragoni, M. Gaspari, D. Guidi, International Journal of Applied Intelligence, 25(2): 159-180, 2006.

3. *NOWHERE - An Open Service Architecture to support Agents and Services within the Semantic Web*
   N. Dragoni, M. Gaspari, D. Guidi, In Proc. of the 2nd Italian Semantic Web Workshop on Semantic Web Applications and Perspectives (SWAP), Trento, Italy, CEUR Workshop Proceedings, 2005.

4. *An ACL for Specifying Fault-Tolerant Protocols*
   N. Dragoni, M. Gaspari, D. Guidi, In Proc. of the 9th AI*IA Conference, Milano, Italy, LNCS, 2005.

5. *Integrating Knowledge-Level Agents in the (Semantic) Web: an Agent-based Open Service Architecture*

   N. Dragoni, M. Gaspari, D. Guidi. In Proc. of the 18th International FLAIRS Conference, AAAI Press, 2005.

6. *A Peer-to-Peer Knowledge Level Open Service Architecture*

   N. Dragoni, M. Gaspari, D. Guidi. In Proc. of WM2005 Workshop on "Peer-to-Peer and Agent Infrastructures for Knowledge Management" (PAIKM05), DFKI, Kaiserslautern, 2005.

Chapter 1

Introduction

Communication protocols have evolved greatly over the last 10 years. Sophisticated Agent Communication Languages emerged from projects such as the Knowledge Sharing Effort or the Foundation for Intelligent Physical Agents (FIPA). More recently, thanks to the ubiquity of the Web, new protocols such as SOAP[66] have been proposed. Such protocols are designed to work specifically with Web Services, and are now evolving to support Semantic Web Services. While software agents have been recognized as one of the key technology to exploit these (Semantic) Web services, programming a set of geographically distributed agents is still a complex task which needs adequate skills and tools to be carried out successfully.

In fact, the realizations of these protocols do not provide a high level abstraction of the communication, so that the communication still subject to low level problems. For example, due to the fact that the communication over the Internet is subject to failures (communication problems, hardware failures, etc), these protocols no longer guarantee the delivery of messages, thus the need for *handling exceptions*. A set of events must be explicitly handled in order to ensure a successfull communication between two entities, be they simple software components such as web services, or more complex entities such as agents.

However, crashed agents and network errors are not managed at high level, but instead using explicit *timeout mechanisms* to ensure that an entity does not endlessly wait for an answer. Following this approach, it is not always clear how to set these timeouts and what action to take when the timeout expires. Furthermore, common concurrency problems such as *deadlock* or *starvation* may arise in the communication and must be explicitly

recognized and handled.

Following these considerations, it seems unlikely that software agents will be developed in the future as Web pages have been created in the past. However, is it possible to reduce this gap? Is it possible to find a programming model which facilitates the development of agents, providing a high level architecture that can automatically manage these problems, at least for a reasonable class of applications?

In this Thesis we present the design and the realization of NOWHERE, an agent platform that represents our answer to this question. The high level approach used in NOWHERE rely in the use of "Knowledge Level" agents, proposed by Newell in 1982[51]. The intuition of Newell is that knowledge is fundamentally different from the symbols used to represent it. Following this intuition, Newell proposed the existence of a new level of system description which he called Knowledge Level and which he located above the symbol (or program) level. The concept of Knowledge Level agents was developed lately by Genesereth and Nilson, who define it in [36] as "a conceptualization of agents in which all excess detail is eliminated. In this abstraction an agent internal state consists entirely of a database of sentences in predicate calculus, and an agent's mental actions are viewed as inferences on this database". In other words, the Knowledge Level rationalizes the agent's behaviour, while the symbol level mechanises the agent's behaviour.

The design and the implementation of NOWHERE continues the work of Prof. Mauro Gaspari and of Dr. Nicola Dragoni regarding Knowledge Level communication in software agents. The research of Gaspari and Dragoni[23, 24], used as starting point for this Thesis, concerns the definition of a Knowledge Level approach to deal with crash failures of agents in open Multi-Agent Systems. The approach is based on $\mathbb{FT}$-$\mathbb{ACL}$, an advanced Knowledge Level Agent Communication Language which allows agents to cooperate in open environments prone to crash failures. While the work of Gaspari and Dragoni led to very interesting results, such as the design of $\mathbb{FT}$-$\mathbb{ACL}$ and its formal specification, a series of questions remain open:

- Is it possible to transform the $\mathbb{FT}$-$\mathbb{ACL}$ specification in a successful language?
- Is it possible to realize a distributed programming mechanism which does not use explicit timeouts to deal with agent failures?

- Is it possible to design a modular runtime support for $\mathbb{FT}$-$\mathbb{ACL}$ which would allow users to easily integrate it in any programming language?
- Is it possible to make all these mechanisms independent from the middleware used for message passing?

These questions represent the main challenges addressed in this Thesis.

Part I

State Of The Art

Chapter 2

Software Agents

The purpose of this Chapter is twofold: to define basic concepts and terminology that are used in this Thesis and to provide a snapshot of the current state of the art, presenting the scenario in which we locate the work done in the context of this Thesis.

2.1 Properties of a Software Agent

The term "software agent", or simply "agent", identifies a concept used in many areas ranging from Multi-Agent Systems to Web services, from Peer to Peer Networks to Grid systems. While the concept of agent is something familiar to a computer scientist, a common definition is still missing, so that agents are defined in almost as many ways as there are commentators in the field.

Analysing the properties that agents may provide, researchers have proposed different classifications for over a decade [55, 60, 34, 73, 42]. Even if these classifications differ, there is some consensus on the features that a software agent should exhibit. In [34] the authors collect a number of different agent's definitions found in the literature and then provide a comprehensive list of agent's properties:

- *Reactivity*. Agents perceive the context in which they operate and react to it appropriately.
- *Autonomy*. Agents have capabilities of task selection, prioritisation, decision-making without human intervention.
- *Pro-activity*. The ability of an agent to be goal-oriented.

- *Communication*. The ability to communicate with other agents, in order to exploit functionality such as cooperation or competition.

- *Adaptability*. It implies sensing the environment and reconfiguring in response. This can be achieved through the choice of alternative problem-solving-rules or algorithms, or through the discovery of problem solving strategies.

- *Temporal continuity*. The agent is a continuously running process.

- *Mobility*. The ability to transport itself from one machine to another.

- *Flexibility*. The ability to have actions that are not scripted.

- *Use of character*. The ability to include believable "personality" and emotional state.

We agree with other researchers that consider the first four features as a set of abilities that a software agent should exhibit in order to be flexible enough to be used extensively in different scenarios [34, 14].

Agents may also provide characteristics that can be considered important only in specific contexts. Mobility - the ability of an agent to stop its execution, move itself (and often its data) to another host and then continue the execution again - is one of these properties. Even if mobility itself does not allow an agent to do something that it is not possible using a static agent, there are many fields where mobile agents can be successfully used, especially when slow network connections are used [37]. At the same time, mobile agents introduce well known security issues that should be treated with care [27], so that it is safer to avoid their use in a scenario with a fast network connection. As a result, the importance of the mobility feature, as well as other properties, is heavily dependent on the context in which agents operate.

Finally, it is interesting to note that having no common definition of an agent is not considered a problem by Russel and Norvig, who state: "the notion of an agent is meant to be a tool for analysing systems, not an absolute characterisation that divides the world into agents and non-agents" [63].

2.2 Multi-Agent Systems

A Multi-Agent System (MAS) (or agent platform, or agent infrastructure) is a system composed of several agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system. Agents may cooperate or they may compete, or some combination of cooperation and competition, but there is some common infrastructure that result in the collection being a 'system', as opposed to simply being a disjoint set of autonomous agents. A Multi-Agent System can be either a closed MAS (the set of agent types is predefined by the entity that sets up and controls the system) or an open MAS (where arbitrary external software agents can join the system).

In the following we focus on open MASs, which are the subject of the work presented in this Thesis. Open MASs are more complex and also more interesting than closed MASs, because they can support "personal" or "user" agents, that carry out tasks automatically for the user.

The amount of agent platforms and prototypes developed in the past few years is extremely vast: far more than one hundred, as stated in [67]. The AgentLink site - European Co-ordination Action for Agent Based Computing - currently provide a list of 129 agent platforms and prototypes. The main reason of so many disjoint efforts is probably because every agent platform is built in order to be used in a predefined context. In fact, depending on the particular context in which the agent is located, some features could become important, or even crucial. Agents architectures are generally well suited for a particular context, the one that the researchers had in mind writing the software. Probably for this same reason the number of available papers presenting comparisons and evaluations between agent platforms is very small. Most of these papers evaluate platforms in a particular context, so the results are not applicable in different fields.

In the last years a huge number of these platform has been abandoned, while communities of researchers started to grow around the most promising platforms. Among others, there are two main approaches used when building a MAS: focusing on communication infrastructure and focusing on the representation of internal agent concepts. The most used approach is probably the one based on communication infrastructure. Here the key point is the use of an Agent Communication Language to exchange knowledge between the agents. Many agent platforms uses this approach, both closed source imple-

mentations (such as Tryllian [15]) and open source ones, such as MadKit [30], Jade [12] and Cougaar [68].

The other approach used in building MAS is to focus on the representation of internal agent concepts, rather than on communication infrastructure. In this case the most used model is the BDI model (Belief - Desire - Intention), conceived by Bratman [13] as a theory of human practical reasoning. Beliefs are informational attitudes of an agent, representing the information that an agent has about the world and about its internal state. Desires (or goals) represent the motivational state of the agent. They consist of objectives or situations that the agent would like to accomplish. Finally, intentions represent the deliberative state of the agent: what the agent chooses to do. Interestingly, there are both MASs focused on ACL that provide a BDI layer, like Jadex [61], and MASs focused on BDI that provide a standard ACL compatibility, like Jack [45].

An important aspect of MAS, however, is still unhandled by all these platforms: the concurrency aspect. Due to the fact that a MAS is composed of a set of agents that act concurrently, a number of problems related to concurrency will arise, such as reliability of agents, synchronisation of competing requests, allocation of resources, physical allocation of agents on the network and so on.

This is exactly the problem addressed by the novel approach provided by our architecture: to provide an infrastructure to handle concurrency issues.

2.3 MAS focused on Communication Infrastructure

In this Section we provide an overview of the state of the art in MASs focused in communication infrastructure, in order to illustrate the scenario in which we locate our architecture. The introduction of the Knowledge Level in agent technology is fundamental because, exploiting this concept, it is possible to manage the knowledge at high level, independently from the programming language used to represent it. Acting at Knowledge Level, agents need a powerful communication system that let them exchange knowledge. Agent Communication Languages (ACLs) have been developed to provide adequate inter-agent communication mechanisms. They allow agents to effectively communicate and exchange knowledge with other agents despite differences in hardware platforms, operating systems, networks and programming languages. In the last decade

many ACLs have been proposed, incorporating specific mechanisms of agent communication. Many of these communication mechanisms are based on the speech act theory, which has originally been developed as a basic model of human communication [64]. In his famous work, "How to do Things with Words" [8], J. L. Austin outlined his theory of speech acts and the concept of performative language, in which *to say something is to do something*. To make the statement *"I promise that p"* (in which *p* is the propositional content of the utterance) is to perform the act of promising as opposed to making a statement that may be judged true or false. Austin creates a clear distinction between *performatives* and *constantives* (statements that attempt to describe reality and can be judged true or false) but he eventually comes to the conclusion that most utterances, at their base, are performative in nature: *"the speaker is nearly always doing something by saying something"*. Speech act theory has been found useful in Multi-Agent Systems as a foundation for communication among agents. In Agent Communication Languages, speech acts are represented as messages expressing *performatives*, i.e., actions which succeed simply because the agent communicates that it is doing so. Thus a message of an ACL is called a *performative*, in that the message is intended to perform some action by virtue of being sent.

In the past 10 years, two important ACLs gained much attention: KQML [31] and FIPA-ACL [32], and both of them adopt the speech act theory. The goal of these languages is to support high-level, human like communication between intelligent agents using Knowledge Level features, so that agents can focus on the use, request and supply of knowledge, without having to deal with symbol level issues. Both KQML and FIPA support high-level agent communication providing a way to encode messages, so that they can be shared by agents coded in different programming languages. Anyway, while these two ACLs share some similarities, KQML and FIPA are still very different, as pointed out in [70].

Interestingly, in the past few years things changed a lot, as more and more researchers started to use FIPA. The result is that nowadays there are no organised efforts to further develop KQML. It happened that the KQML effort was pretty much subsumed by FIPA's activities. Many of the people that developed KQML worked with FIPA beginning in the late 90s. In the current scenario, KQML is considered a death project, while FIPA is now an IEEE standards effort. The same thing that happened to the communication language (KQML vs FIPA), happened also to KQML-compliant MAS and

FIPA-compliant MAS. Even if KQML is not developed anymore, it is still possible to download the KQML API, which provide primitives to develop KQML-enabled agents. While many KQML implementations can be found in the original KQML software page (`http://www.cs.umbc.edu/kqml/software/`), only one of this seems to be downloadable: a 1998 version of a C and lisp implementation. Other KQML-compatible agent platforms can be found: AgentBuilder [4] and Jack [45]. However it is still difficult to work with these platforms, because, being closed source, it is not possible to have enough details about their implementation. The fact that FIPA is currently by far the most used ACL, however, does not imply that it represents the best approach for agent communication. This statement is supported by the fact that Multi-Agent Systems researchers started to focus on issues other than communication, often using an ad-hoc communication language and infrastructure in any implementations. In fact the problem of agent communication is only one of the problems involved in the creation of a concurrent system.

2.3.1 JADE Agent Platform

Jade [11] is currently one of the most used agent platform both in academy and in the industry. It was developed jointly by CSELT (Centro Studi e Laboratori Telecomunicazioni) in conjunction with the Computer Engineering Group of the University of Parma. While a FIPA sponsored platform, FIPA-OS [54, 14], was built in order to create a FIPA-ACL standard platform, soon JADE took its place. Now that the FIPA-OS project is halted, JADE is considered one of the most complete FIPA implementations.

JADE is a full FIPA complaint platform, written in the Java programming language, which include the following standard FIPA components [7, 5]:

- *The Agent Platform (AP)*. It provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents.

- *The Directory Facilitator (DF)*. It is an optional component of the AP that provides yellow pages services to other agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents.

- *The Agent Management System (AMS)*. It is a mandatory component of the AP. The AMS exerts supervisory control over access to and use of the AP. The AMS maintains a directory of agents registered with the AP, providing white pages services to other agents.

- *The Message Transport Service (MTS)*. This is the default communication method between agents on different APs.

Due to the fact that the JADE project started several years ago, it does not directly support some of the current major key technologies, such as Web services, and inter-agent communication with firewall avoidance. However many third party extensions can be downloaded in order to overcome some limitations.

From the point of view of the developer, JADE offers several features:

- A FIPA-compliant Agent Platform, written in the Java programming language.

- Distributed Agent Platform. The agent platform can be split on several hosts and only one Java Virtual Machine is executed on each node. Agents are implemented as Java threads and suitable transport is chosen for message delivery, depending upon relative location of sender and receiver agents.

- Multithreaded execution environment.

- Object Oriented programming environment. Most concepts present in FIPA specifications are represented as Java classes, so that a uniform programming interface is provided to developers.

- Library of interaction protocols. Ready to use behaviour objects are provided for the standard interaction protocols such as FIPA-REQUEST and FIPA-CONTRACT-NET. To build an agent that can act according to an interaction protocol, application developers just need to implement domain specific actions, while all application independent protocol logic will be carried out by JADE framework.

- Administration GUI. Common platform management operations can be performed through a graphical user interface, showing active agents and agent containers. Using this GUI, platform administrators can create, destroy, suspend and resume agents, besides creating domain hierarchies with multiple federated DF agents.

JADE does not directly support other programming languages other than Java, but it should be compatible with any FIPA-compliant platforms. Regarding the physical network layer, agents running on the JADE platform send messages using a generic `send` call method. However, the internals of JADE select the most appropriate transport protocol for each different situation:

- If the receiver agent lives in the same agent container (that is, agents running on the same Java Virtual Machine), the Java object representing the ACL message is passed to the receiver by using an event object, without any message translation.
- If the receiver agent lives in the same JADE platform but within a different container, the ACL message is sent by using Java Remote Method Invocation. Java RMI allows transparent object marshaling and unmarshaling, avoiding tedious message conversions. Apart from performance, the agent receives a Java object, just like intra-container messaging.
- If the receiver lives on a different agent platform, the IIOP protocol and OMG IDL interface are used, according to the FIPA standard. This involves translating ACL message object into a character string and then performing a remote invocation using IIOP as middleware protocol. On the receiver side, an IIOP unmarshaling will occur, yielding a Java String object, which will be parsed into an ACL message object. Eventually, the Java object will be dispatched to the receiver agent (via Java events or RMI calls).

2.3.2 Open Agent Architecture

The Open Agent Architecture (OAA) [17] is a Multi-Agent System that focuses on enabling more flexible interactions among a dynamic community of heterogeneous software agents. OAA supports several programming languages using a Facilitator, a specialized server agent that coordinates the activities of agents for the purpose of achieving higher-level, complex problem-solving objectives. Instead of using FIPA ACL or KQML, OAA adopts its own communication language, called Interagent Communication language (ICL). ICL is based on an extension of the Prolog language, and uses the Prolog syntax. ICL provides primitives to achieve what the authors call “delegated computing”:

instead of each agent hard-coding its interactions (method calls or messages), explaining *how* and *who* it will interact with, OAA agents express interactions in terms of needs delegated to a Facilitator agent

JADE and other similar platforms provide two basic features:

- The architecture provides a service repository containing interface specifications for available services, the Directory Facilitator
- When an agent requires the service of another, it queries the repository to find a service by specified name, ID (identifier), attribute, and then interacts with the agent under control of it's own code. The Requesting agent decides which agents it will interact with and how the interactions will occur, and is thus responsible for choosing, monitoring and maintaining the interaction session.

In OAA, agents use the ICL to register their capabilities with a Facilitator. Competences of other agents are then exploited using ICL, asking one or more agents for the solution of a particular goal, written as a Prolog-style declaration, such as *send(email, Person, Message, AdditionalParameters)*. The requested capability is then matched against the set of all the provided capabilities using the unification algorithm. The key difference is that the OAA architecture takes care of the process of choosing, monitoring and interacting with proper agents that provide this service.

While the publish/request of capabilities is heavily simplified by the OAA architecture, failures of agents are not managed. For example, if a communication error arises using the Java OAA version, a Java exception is raised and the agent must treat the exception.

One weakness of the OAA architecture is that it is not actively developed anymore. Even if several papers describe enhanced OAA prototypes with Web services and Semantic Web integration, the latest version does not provide support for these technologies.

2.3.3 Other Platforms

Just like FIPA-ACL gained attention and became the de-facto ACL despite KQML, many agent platforms were abandoned in favour of JADE. There are Java-based prototypes written to test interesting approaches, like Bee-gent [16] that, as opposed to other systems which make only some use of agents, completely "agentifies" the communication that

takes place between software applications. Aglets [47] is a platform based on the concept of an "aglet", a Java agent able to autonomously and spontaneously move from one host to another. The Java aglet extends the model of network mobile code made famous by Java applets. Like an applet, the aglet can migrate across a network, but it is also able to carry its state.

Many other platforms are compliant to the FIPA specifications. April [49] (Agent PRocess Interaction Language) is interesting because it is not a platform, but instead a strongly typed, process oriented symbolic language implemented in C, for developing Multi Agent Systems compatible with the FIPA standard. Other projects are very similar to JADE, FIPA-compliant platforms written in the same programming language, such as Grasshopper [10], Zeus [56], OpenCybele [44] and Cougaar [43]. The Cougaar project, supported by DARPA, is the most interesting one. It is focused on the plugin technology in order to create an extensible platform that supports also some primitives for planning and execution.

2.4 Towards Knowledge Level in MAS

As Gaspari discusses in [35], the notion of Knowledge Level agent cannot be straightforwardly extended to a distributed framework such as MASs, because problems concerning communication and concurrency may arise when several Knowledge Level agents interact (for example deadlock or starvation). In particular, a common agreement on what Knowledge Level programming means in MAS is still missing. FIPA and KQML, in fact, do not contain primitives that consider concurrency issues, so that the developer is forced to handle three kinds of potential communication problems:

1. to manually avoid or to manage concurrency communication problems like deadlock or starvation;

2. to manually handle agents faults;

3. to manually manage low level network problems, such as network latency, agents temporarily disconnected from the network and so on.

In the same paper, Gaspari postulates a set of requirements that an agent communication language should satisfy to be regarded as Knowledge Level. The requirements are:

1. The programmer should not have to handle physical addresses of agents explicitly.
2. The programmer should not have to handle communication faults explicitly.
3. The programmer should not have to handle starvation issues explicitly. A situation of starvation arises when an agents performative never gets executed despite being enabled.
4. The programmer should not have to handle communication deadlocks explicitly. A communication deadlock situation occurs when two agents try to communicate, but they do not succeed; for instance because they mutually wait for each other to answer a query.

The work done in the context of this Thesis follows this approach, providing a realization of the Fault Tolerant Agent Communication Language $\mathbb{FT}$-$\mathbb{ACL}$[23, 26, 25]). $\mathbb{FT}$-$\mathbb{ACL}$ adopts asynchronous non-blocking primitives together with success and failure continuations to provide a framework where Knowedge Level agents can interact.

Part II

The NOWHERE Architecture

Chapter 3

Introduction to NOWHERE

In this Chapter we present an introduction to the architecture built in the context of this Thesis. First we provide an overall description of the architecture, followed by a brief comparison with other platforms and by a list of its main features. Then in the last part we describe the internal structure, showing the three levels that compose the architecture.

3.1 Overall description

The most important concept behind NOWHERE is the idea to create an infrastructure for Knowledge Level agents. The key component of this architecture is then the agent communication language, FT-ACL. Around the implementation of the FT-ACL, which provides high level primitives to agents, we built an infrastructure where agents can interact.

Designing the architecture we tried to create a kind of "generic" platform, which can optionally be extended in order to satisfy specific requirements. We achieved this goal in two different ways. Firstly, splitting a single agent in two components: one that provides a set of facilities and another one that runs the agent code, exploiting these facilities. Using this technique, we are able to easily adapt NOWHERE to different programming languages, enabling interoperation between different agents. Secondly, creating the infrastructure using three layers that communicate using standard interfaces. Using layers it is possible to change part of the architecture, maintaining compatibility with the rest of the platform. For example it is possible to change the layer used to send messages, in order to adapt NOWHERE to different scenarios.

A general view of our architecture is shown in Figure 3.1.

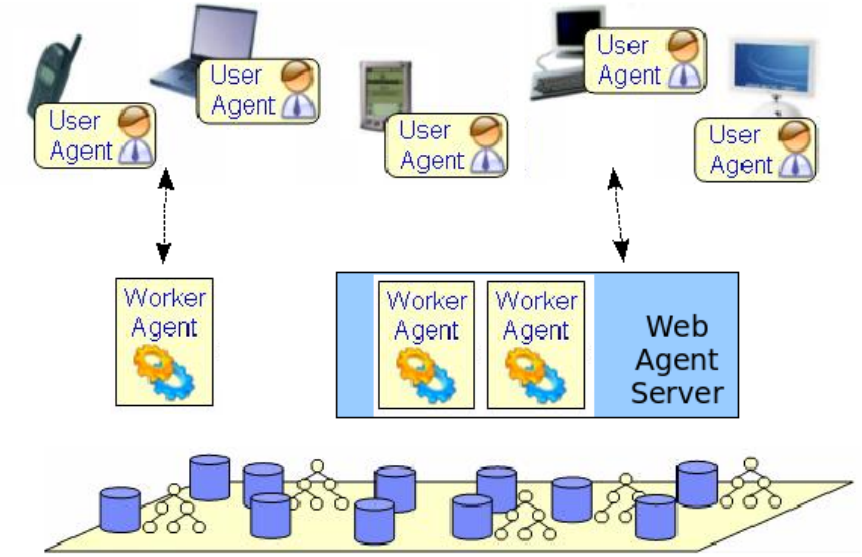

Figure 3.1: NOWHERE Architecture

NOWHERE supports both *User agents* and *Worker agents*. User agents act as interfaces between users and the Web, providing support for discovering and invoking services. Users can configure their User agents with their preferences. They can be always connected to the MAS or they can disconnect themselves when their users want. Worker agents are able to retrieve, execute and compose services provided by other agents in order to create more sophisticated services. Contrary to User agents, Worker agents are always connected to the MAS and act like daemon processes. NOWHERE provides full integration between agents and Web services: agents' capabilities can be exported as Web services and existing Web services can be "agentified" by a virtual agent. This virtual agent invokes the Web service and manages its reply according to the requests made from other agents. In this way the virtual agent acts as a wrapper, exporting the Web service as its capability to other agents. Web service integration is achieved using Web Agent Servers, which extend a Web Server with agents' functionality. Web Agent Servers are geographically distributed (as Web Servers are) and provide a set of Web services to the outside world which represents their capabilities. This set can dynamically change because of new publication of Web services and/or modification of the existing ones. Worker agents can also be used to provide information retrieved from standard sources (like databases or the Web) to User agents in a more structured way, publishing advanced

capabilities to User agents.

Implementing a complex project like an agent platform became more easy when using open source software. NOWHERE is released as open source and contains many external open source code, such as the standard Base64 encoding protocol, the SHA-1 algorithm and the whole infrastructure used to send messages (for example Jabber or JXTA).

3.2 Comparison with other platforms

The NOWHERE platform shares many features with other agent architectures. Probably the OAA architecture is the most similar platform. They share the concept of a Facilitator component that provide complex functionality to agents. However, while the OAA's Facilitator is a shared component that can manage many agents, in NOWHERE every agent has its own Facilitator. Moreover, the NOWHERE Facilitator contains a failure detector to handle agent crashes. The mechanism used to register capabilities is also very similar, with the same callback system: the agent defines a function code to handle the reply of a previous request that will be automatically called by the architecture. Again, NOWHERE supports also communication failures using a second callback mechanism. Finally, the OAA's "delegating computing" concept is very similar to the NOWHERE's anonymous interaction mechanism. In both cases an agent is able to request a specific service from a set of agents, without any prior knowledge about their names or their locations.

The main difference with OAA is that NOWHERE uses an ACL based on the speech acts theory, where communication performatives provides a Knowledge Level layer. Also, current standard technologies such as Web services are not integrated in the current version of OAA.

On the other hand, NOWHERE adopts a plugin methodology similar to the one utilized in JADE. Plugins are used to change specific functionality, such as the method used to send inter-agent messages. NOWHERE extends this feature, providing an architecture built on three different layers that can easily be extended. Moreover, NOWHERE takes care about current technologies, such as Web services, an aspect already present in JADE, where these extensions are mostly implemented by third party developers.

3.3 Main Features

In this Section we present a list of the main features of our architecture. This list can be helpful when comparing NOWHERE with the huge number of agent platforms available.

A Dynamic and Open Architecture for User agents.

In the research on Multi-Agent Systems there is an increasing emphasis on the open-ended nature of agent systems, which refers to the feature of allowing the dynamic integration of new agents into an existing agent system. This feature becomes particularly relevant when agents are developed on the Web, where they are usually implemented by different people at different times. NOWHERE allows new agents to dynamically connect themselves to the system, providing new capabilities to other agents.

User agents are a special kind of software utilised by computer users. A User agent is an autonomous entity that acts using a specified set of rules set by its owner, and that makes choices suitable to reach a predefined goal. While every agent platform can theoretically be used to program User agents, an additional set of features is essential, for example the ability to bypass firewall or the ability to easily execute Web services.

A platform for Knowledge Level agents.

One of the main differences between NOWHERE and other agent platforms is the support for Knowledge Level (KL) agents.

Using a Fault Tolerant Agent Communication Language, $\mathbb{FT}$-$\mathbb{ACL}$, the programmer does not have to explicitly handle many low level issues, such as network and concurrency related problems. The support for KL agents is realised by two components:

- The **Facilitator**, written in the Java language. This component provides high level primitives for sending and receiving messages, using a fault tolerant architecture.
- The **Agent Dispatcher** component. This component provides the basic functionality to support communication between KL agents and the Facilitator. It can easily be implemented in (virtually) any language that provide tcp support.

Using two separate components has several advantages. First of all the developer is free to choose the preferred language for the agent code, while the whole communication is

always handled by the Facilitator. Agents written in different programming languages can then talk each other, using the ACL primitives. Furthermore, the agent and the Facilitator could also run on different machines: for example a mobile phone, with a limited computational power, could run only the agent code, using the Facilitator hosted on a local computer. The idea is to keep all the communication issues, such as the fault-tolerant behaviour, in the Facilitator.

Integration with Web services.

Software agents running in NOWHERE can share capabilities with other agents. Capabilities are described using a subset of WSDL [18], the same language used to describe Web services. Web services integration is achieved in two ways:

1. NOWHERE provides a functionality which allows agents to register existing Web services. As soon as a Web service is registered it becomes reachable as a *virtual agent* and it can be transparently invoked using $\mathbb{FT}$-$\mathbb{ACL}$ primitives.

2. In a complementary way, an agent could register a particular competence as a Web service, so other programs can interact with the agent without sharing the agent architecture. The newly generated Web service is hosted in the external Web Agent Server.

Support for Interoperability.

NOWHERE agents can be realised in any programming language including AI languages or knowledge representation languages, provided that they react to a well defined protocol based on the standard primitives of $\mathbb{FT}$-$\mathbb{ACL}$. Although emerging standards for the Web use formalisms based on XML, most of AI systems are still being developed using specific AI technologies and languages which usually are not compliant with Web standards, but provide powerful engines and a rich set of libraries. From a practical point of view it is not feasible to translate all these technologies in XML based formalisms or to commit to a single programming language. Thus, enabling the integration of agents written in different programming languages is essential to a large scale exploitation of Knowledge Level agents on the Web. Interoperability is also guaranteed by the fact that NOWHERE is an open source project, that can be freely used or mod-

ified. A Java and a Python versions will be soon available on the popular sourceforge (`http://www.sourceforge.net`) site.

3.4 NOWHERE's Internal Structure

NOWHERE is composed of three interconnected layers: the Knowledge Level at the top, the Architecture-Level in the middle and the Network-Level at the bottom, as shown in Figure 3.2

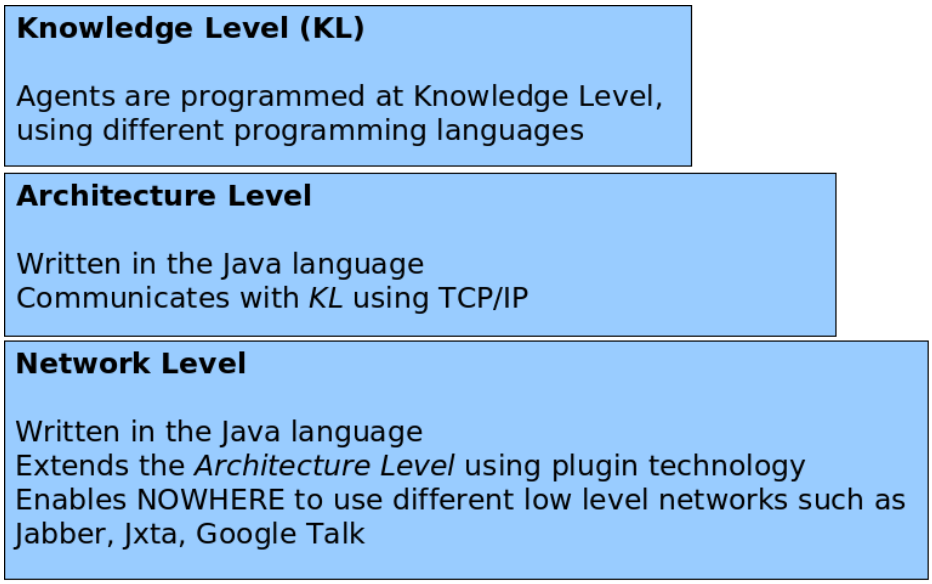

Figure 3.2: NOWHERE Layers

3.4.1 Knowledge Level Layer

The Knowledge Level layer contains the set of high level primitives used by agents to interact with the MAS. In order to enable inter-agent communication between two agents written in different languages, it is sufficient that both languages provide the Knowledge Level layer. This facilitates the porting of NOWHERE to other programming languages, because the lower levels must not be rewritten in the new programming language. From the point of view of the developer, the Knowledge Level layer is a *stub* that must be

extended in order to create Knowledge Level agents. At this time we provide Knowledge Level support for Java and Python.

3.4.2 Architecture Layer

The Architecture Layer consists of the Facilitator component. Knowledge Level layers written in different programming languages communicate with different instances of the same Java Facilitator, using the TCP protocol. These two layers can also be splitted in different computers. For example, it is also possible to keep the Knowledge Layer on a device with low computational power, such as a mobile phone or a handheld device, exploiting the Architecture Layer provided by a desktop computer or a server. Crashes of agents are detected at the Architecture-Level using various countdown timers, ensuring that an agent will not endlessly wait for a reply. This middle level contains also the algorithm used to manages messages, associating requests to their responses and vice-versa.

3.4.3 Network Layer

The bottom level, the Network Layer, manages the physical route of the messages, providing one-to-one and one-to-many communication primitives and supporting the creation of groups of agents that are interested in the same topic. Using different Network-Levels as plugins, NOWHERE can be adapted to very different scenarios. Currently we provide a Jabber Network Layer and a JXTA Network Layer. Using the Jabber Network Layer it is possible to exploit the Jabber protocol (or the Google Talk protocol) to send and receive messages. Due to the fact that the Jabber network follows a client/server model, the resulting architecture will be very fast, providing support for agents with realtime properties. On the other hand, using the decentralised JXTA Network Layer, the resulting architecture will provide a better scalability, with more latency in the communication. Of course other Network Layer plugins can be added, such as ad-hoc ones, exploiting a standard Java interface that links the Architecture Layer to the Network Layer.

Chapter 4

NOWHERE's Agent Communication Language

NOWHERE is designed to support various kinds of communication primitives based on FT-ACL. The base language that we designed is the *core language*, that supports asynchronous, non-blocking primitives. We also defined an *extended language* that uses blocking primitives to provide support for programming sequential agents like proactive agents. In the following Sections we describe the key ideas behind these languages. The full specification of the core language is then presented in Section 4.2.

4.1 FT-ACL: A Fault Tolerant Agent Communication Language

FT-ACL is the communication language implemented in the NOWHERE platform. Like other popular ACLs such as FIPA ACL and KQML, FT-ACL is based on the speech acts theory. However the expressive power of these languages is very different.

FIPA ACL sends every communication performative as content of asynchronous message passing. More precisely: the concurrent semantics is the same for every performative. FIPA ACL performatives are encoded in an `ACLMessage` object, which is then transmitted in the queue associated to the recipient agent using a *send* primitive. The same send primitive is used by FIPA ACL to transmit every performative.

Instead, in KQML different communication performatives have different declarative semantics. For example the performative *insert(A, B, p)* (Agent A wants B to insert p in his knowledge base) leads to the postconditions:

- *know(A, bel(B,p))* (Agent A knows that p is in B's knowledge base) and

- *bel(B,p)* (p is in B's knowledge base).

This insert performative cannot be realized just sending a message like in the FIPA model, but it is necessary that A receives an acknowledge message about the insertion of p in B's knowledge base. Moreover, in order to achieve Knowledge Level programming, agent A must also manage the situation in which agent B crashes without sending back the acknowledge message.

$\mathbb{FT}$-$\mathbb{ACL}$ takes into account these problems providing an ACL for Knowledge Level agents. It consist of a set of performatives, each one with a different concurrent semantics. Every performative consist of a complex behaviour that is fundamentally different from a simple send primitive. $\mathbb{FT}$-$\mathbb{ACL}$ provides support for one to one primitives as well as one to many primitives. One to many primitives can also be used to realize the anonymous interaction mechanism, where an agent asks a set of other agents for a specified capability, without any prior knowledge about their names or their locations.

$\mathbb{FT}$-$\mathbb{ACL}$ does not dictate any representation language to be used as content of the messages, so that the developer is free to choose the one that best suits its needs. Finally, $\mathbb{FT}$-$\mathbb{ACL}$ deals with failures of agents, adopting the model specified in the following.

4.1.1 Failure model

Following a well known classification of process failures in distributed systems [50], we say that an agent is *faulty* in an execution if its behaviour deviates from that coded in the algorithm it is running; otherwise, it is *correct*. A faulty agent *crashes* if it stops prematurely and does nothing from that point on. $\mathbb{FT}$-$\mathbb{ACL}$ manages faults considering only crash failures. This is a common fault assumption in distributed systems, since several mechanisms can be used to detect more severe failures and to force a crash in case of detection. $\mathbb{FT}$-$\mathbb{ACL}$ deals with crash failures of agents allowing the programmer to choose what actions to invoke for each interaction they perform in the MAS. For example, an agent could decide to ignore the crash of another agent for a certain interaction while it could decide to take some precise actions if the same agent crashes in another more critical interaction.

4.1.2 The Core Language

The core language is a set of speech acts performatives implemented with asynchronous, non-blocking primitives. Using non-blocking primitives, the agent is able to continue the execution of a task without have to wait for the reply of the sent message. In other words, the use of non-blocking primitives means that, when executing a communication primitive, the control flow always passes to the next instruction, even if the recipient agent has crashed. This is a sound behaviour because in asynchronous systems, when a communication action is executed, it is not always possible to detect if the recipient agent has crashed. FT-ACL allows the programmer to deal with faulty agents providing a high level mechanism which binds specific success and failure continuations to communication primitives. Failure continuations are optional, but they should be specified to deal with a possible failure, so that if the recipient agent fails the failure continuation is executed.

In a similar way, the success continuation is called if the communication succeeds and the reply is received. To attach continuations in communication primitive is particularly useful when programming reactive agents using functional programming languages such as Lisp or Python, or logic programming languages such as Prolog. Continuations are automatically executed based on success or failure conditions. While using continuations may look strange to imperative or object oriented language programmers (such as Java programmers) it is very common to use them when developing concurrent software.

The following Python-like pseudo code describes a fault tolerant version of a sample `performativeName` primitive, like for example `askOne`, illustrating how FT-ACL continuations work.

```
1  def mainCode():
2      ...  some code ...
3      performativeName(recipientAgent, content, onAnswer, onFail)
4      ...  other code ...
5
6  def onAnswer(replyMessage):
7      % Here we handle the success continuation
8      % of the performativeName primitive
9
10 def onFail():
11     % Here we handle the failure continuation
12     % of the performativeName primitive
```

In the code presented above there is a main function (`mainCode`, lines 1-4) that at some point sends a message to the agent `recipientAgent` using a generic performative `performativeName` (line 3). A typical send primitive is usually realised using only two arguments: the recipient (`recipientAgent`) and the content of the message that must be sent (`content`). Instead, using the $\mathbb{FT}$-$\mathbb{ACL}$ style, the primitive includes also the success and the failure continuation, `onAnswer` and `onFail` respectively. These parameters are functions that will manage the success and the failure continuation of this specific communication primitive.

Due to the fact that the core language uses non-blocking primitives, after the execution of `performativeName`, the control flow immediately passes to the next instructions, contained in the "`... other code ...`" block, line 4. When the reply message is received, the success continuation `onAnswer` (lines 6-8) is executed, with the parameter `replyMessage` instantiated with the received reply message. Otherwise, if a communication error arises, then the failure continuation `onFail` (lines 10-12) will be executed.

Agents written using the core language are easy to program because a set of Knowledge Level properties (that we recall in the following) holds:

(1) The programmer does not have to manage physical addresses of agents explicitly.

(2) The programmer does not have to handle communication faults explicitly.

(3) Communication is Starvation free.

(4) Communication is Deadlock free.

4.1.3 The Extended Language

We have also defined an extended language, useful when programming proactive agents in imperative languages languages such as Java. While the design of such language is not completed yet, we present here the key idea. The extended language extends the core language providing blocking primitives with a syntax similar to the exception handling mechanism used in programming languages, such as the Java's try & catch statement. An example of the `performativeName` sample communication primitive using the extended language is given in the following.

```
1  def mainCode():
2      try:
3          ...  some code ...
4          replyMessage = performativeName(recipientAgent, message)
5          ...  other code ...
6      except communicationError:
7          % Here we handle exceptions
```

This new version of the `performativeName` primitive uses just only two parameters: the recipient agent and the message that must be sent. After sending the message, a reply is waited and the variable `replyMessage` (line 4) is then instantiated with its value. Communication errors are considered in the exception block (lines 6-7).

Using the extended language, however, KL-properties do not hold. This weakness resides in the use of blocking primitives, that limit the concurrent behaviour. While the extended language can be useful in some cases, especially when programming sequential proactive agents, the developer must explicitly handle concurrent aspects of the agents, such as concurrent access to its internal resources.

4.2 Core Language Primitives

Core language primitives support communication providing agents with the capability to exchange messages and invoke services provided by other agents. A `Message` object encapsulates the content of the communication in a language-independent way, so that

agents written in different languages are able to exchange messages. Using the FT-ACL communication language, agents provide simple or complex capabilities to other agents through services. These services are described using a subset of WSDL [18], the standard XML format for Web services. Services differ from messages because they have a description that holds information about several aspects, including the name of the service, its parameters and the data types used. They are used with specific primitives such as `askOne`, `askEverybody` and `tell`.

In the NOWHERE architecture, a service description is contained in a `Description` object. To manage the invocation and to send the reply of a service, NOWHERE provides a `Request` and a `Response` object, that can be retrieved from `Description`. Both the `Request` and the `Response` objects are templates containing relevant information extracted from the service description, such as the name of the parameters of the service. In order to invoke a service (to provide a response), a `Request` (a `Response`) template must first be filled in with the correct information. Due to the fact that these templates contain part of the service description, they simplify the actions of invoking and replying to a service. A `Message` object is very similar to a `Request` or a `Response` but is more generic, because it can contain every kind of data, while the other two objects can only contain the data specified in the service description. For this reason the NOWHERE architecture is designed to send only `Message` data types, so that `Request` and `Response` must be codified (decoded) into (from) messages (a detailed description is given in Section 5.2.2).

The primitives specified by the core language, that follow the FT-ACL specification, are presented in Table 4.1. In the following we describe them in details, providing also a few examples. Due to the fact that the NOWHERE architecture supports many programming languages, we provide code examples which illustrate the various primitives for both Python and Java, two of the languages already supported by NOWHERE.

4.2.1 One-to-one knowledge exchange and message handling

Communication between two agents can be achieved using the `inform` primitive, the very basic communication method provided by NOWHERE. The syntax of this primitive is:

```
inform(recipientAgent, message)
```

One-to-one knowledge exchange
`inform(recipientAgent, message)`
`informACK(recipientAgent, message, onAnswer[, onFail])`
Using functions to manage specific messages
`handler(message, function)`
Managing Services
`Description loadDescription(WSDL_Description)`
`Description makeDescription(targetNS, operation, parameters, returnParameters)`
Using functions to manage specific services
`handler(request, function)`
Providing and Requesting services
`askOne(recipientAgent, request, onAnswer[, onFail])`
`tell(recipientAgent, response)`
Service publishing
`register(description)`
Anonymous service request
`askEverybody(request, onAnswer[, onFail])`
`allAnswers()`

Table 4.1: Core Language Primitives

where `recipientAgent` is the unique ID (identifier) of the recipient agent and `message` represents the message containing the information to be sent. The `inform` primitive is used to send a message to another agent, without any feedback about the delivery status. No actions are performed by the sender agent if the recipient receives the message, as well as no actions are performed if the message is not delivered for some reason.

`informACK` is a similar but more powerful primitive. With this primitive, a success and a failure continuation are defined, specifying an action to take if the message is delivered and an optional action to take if the message does not reach the recipient for come reason. The syntax is:

```
informACK(recipientAgent, message, onAnswer[, onFail])
```

where the `recipientAgent` and the `message` parameters are the same of the previous `inform` primitive. The `onAnswer` parameter represents the function to be called if the message is delivered (success continuation) while the optional `onFail` parameter represents the function to be called if the message is not delivered (failure continuation).

In order to illustrate how this primitive works, let us introduce a simple scenario where agent A must send the knowledge about a new assertion to agent B. The Python-like code for these agents can be found in figures 4.1 and 4.2.

```
1  msg = message('new_assertion')
2  msg.setElement('a_new_assertion', 'assertionContent')
3  self.informACK(agentB, msg, messageReceived, errorOccurred)
4
5  def messageReceived():
6      print 'Agent B has received the message'
7
8  def errorOccurred():
9      print 'Error:  message not delivered!'
```

Figure 4.1: Code of Agent A - Python

Lines 1-2 of Figure 4.1 show the creation of a message with the name "new_assertion". The `Message` data type contains an element with name 'a_new_assertion' and value represented by the string "assertionContent". The primitive `informACK` is contained in line

3, where the success continuation is bound to the function `messageReceived` (lines 5-6) and the failure continuation is bound to the function `errorOccurred` (lines 8-9). One of these two functions will be run as the result of the communication primitive. The Figure 4.2 presents the code of agent B. The `Dispatcher` function found in line 1 is a standard function that can be implemented by Knowledge Level agents in NOWHERE. It will be automatically called by the architecture runtime support when an incoming message is received. Every actions that an agent want to take in response to a specific message, can be encoded in this function (lines 2-3).

```
1  def dispatcher(self, m):
2      if m.getName() == 'new_assertion':
3          # Appropriate actions are taken
```

Figure 4.2: Code of Agent B - Python

Incoming messages can also be managed using the `handler` primitive. Figure 4.3 illustrates the code for an equivalent agent B that uses the `handler` primitive to manage the same incoming message.

```
1  msg = message('new_assertion')
2  handler(msg, assertionHandling)
3
4  def assertionHandling(m):
5      # Handling assertion messages
```

Figure 4.3: Code of Agent B - Using the Handler Primitive - Python

Using the `handler` primitive, the developer specifies a function that will manage a certain set of messages. In line 1 a new message is created with the name "new_assertion". The second line states that each incoming message that match the one we just defined will be handled using the `assertionHandling` function (defined in lines 4-5). In this example every incoming message with name "new_assertion" will be handled by the specified function. Other elements can be specified in the message created in line 1, obtaining a more restrictive set of messages to be handled: the system uses a pattern matching algorithm to match the property of the specified message with the incoming message.

4.2.2 Request/Response Performatives

In order to use services, a `Description` object (that stores the data about the service) must first be obtained. Such descriptions can be retrieved in two ways: from an existing WSDL file or from an explicit user specification. The `loadDescription` primitive can be used to parse a WSDL file either from a local resource or from the Web, retrieving a `Description` object that can then be used in NOWHERE. The `loadDescription` primitive has the following syntax:

```
Description loadDescription(WSDL_Description)
```

where `WSDL_Description` is a string containing the file or the resource to parse.

If a WSDL file does not exist, the developer can create a new WSDL file from scratch or, in alternative, use an explicit specification. The `makeDescription` primitive automatically creates a `Description` object instantiated using the parameters provided by the user, using the following syntax:

```
Description makeDescription(targetNS, operation,
   parameters, returnParameters)
```

The parameters are, respectively, the target namespace of the service, the name of the operation that we are describing, the definition of its parameters (name and data type) and its return values (name and data type). From every `Description` object it is possible to retrieve a `Request` and a `Response` object that must be used in order to invoke a service and to send the results of a provided service. A detailed description of these objects is given in Section 5.2.2

Using functions to manage specific services.

Just like we have already seen for messages, it is possible to define a function that will take care of a specific kind of services. This function can be specified using the `handler` primitive, this time supplying a `Request` object as first parameter:

```
handler(request, function)
```

Every attribute specified in the `request` template is matched with the incoming message. It is then possible to filter services with specific values for certain parameters, using a pattern matching behaviour. A fragment of a code that highlight this primitive is presented in Figure 4.4.

```
1  serviceDescription = loadDescription(wsdlFile)
2  request = serviceDescription.getRequest()
3  request.setParameter(''parameter1'', ''thisValue'')
4  handler(request, serviceHandler)
5
6  def serviceHandler(m):
7      # Handling service requests
```

Figure 4.4: Managing Specific Services using a Specific Function - Java

In this example, a description of a service is first loaded from a file and a `Request` object is then retrieved from it (lines 1-2). Here we assume a service with just one parameter, `parameter1`, associated to a string value. The value "thisValue" is specified for `parameter1` in line 3 and the resulting `Request` object is then bound with the function `serviceHandler` (line 4). As a result, this function will receive every incoming request for the loaded service that have "thisValue" as value for the parameter `parameter1`.

Providing and Requesting Services.

The `askOne` primitive must be used to invoke a service provided by another agent. The syntax is:

```
askOne(recipientAgent, Request, onAnswer[, onFail])
```

To illustrate the `askOne` primitive, let us introduce another simple scenario, where the Reader agent provides a services to retrieve the temperature. The Collector agent acts as a manager, asking Reader agent for its temperature. In this scenario we use just one service: `temperature`, with no parameters and two return values: `myTemperature` and `myLocation` that stores respectively the temperature and the associated location of the specific measurement given by the Reader agent. The code of the Reader agent is presented in Figure 4.5 while Figure 4.6 shows the code of the Collector agent.

```
1  Description description = loadDescription(WSDLFile);
2  Request request = description.getRequest();
3  handler(request, "provideTemp");
4
5  public void provideTemp(Message m) {
6      Response response = description.getResponse();
7      response.setParamater(``myLocation'', "laboratory");
8      response.setParameter(``myTemperature'', ``32'');
9      tell(m.getSender(), response);
10 }
```

Figure 4.5: Code of the Reader Agent - Java

The first line of code, containing the `loadDescription` primitive, is shared by the two agents and it is used to retrieve the `Description` object of the service. This object is exploited by the Reader agent (line 2 of Figure 4.5) to create a `Request` object. In order to provide this service, the Reader agent binds the `Request` object with the function `provideTemp` (line 3). Without specifying fixed valued for the parameters of the service, every request concerning this service will be managed by the `provideTemp` function (lines 5-10), that will send back a `Response` object with the parameters `myLocation` and `myTemperature` properly instantiated. The reply of a service must be send with the `tell` performative:

```
tell(recipientAgent, response)
```

The code of the Collector agent (Figure 4.6) is composed of a main part (lines 1-3), which contains the service invocation, and by the two functions that will manage the continuations: `printTemp` (lines 5-8) and `fail` (lines 10-11). The `printTemp` function accepts a `Message` parameter that represents the Reader agent's reply. If the Collector agent receives a reply, it extracts the `Response` object using the `retrieveResponse-FromMessage` function (line 6). The result of the executed service is then printed on the screen (lines 7-8).

```
1  description = loadDescription(WSDLFile)
2  request = description.getRequest()
3  self.askOne(agentA, request, printTemp, fail)
4
5  def printTemp(msg):
6      response = description.retrieveResponseFromMessage(msg)
7      print 'Temperature in ', response.getParameter('myLocation')
8      print ' is:  ',response.getParameter('myTemperature')
9
10 def fail():
11     print 'Agent not found!'
```

Figure 4.6: Code of the Collector Agent - Python

4.2.3 The Anonymous interaction mechanism

NOWHERE gives support for invoking a service from a set of agents. This mechanism is also called content-based request, because a service can be invoked specifying its content, without have to specify the names of the agents that provide it. In order to use this feature, agents must first publish their services using the `register` primitive. The syntax of this primitive is:

```
register(description)
```

The `register` primitive must be invoked after the associated `handler` primitive: a function to manage this service must already be defined. Published services can be invoked using the `askEverybody` primitive, whose syntax is:

```
askEverybody(Request, onAnswer, onFail)
```

The parameters are the same of the `askOne` primitive seen before, except that in this case the recipient agent is not specified. It is the runtime support that will send the request to all (and only) the agents that provide the wanted service. Following the same scenario introduced for the `askOne` primitive, figures 4.7 and 4.8 present the source code for the Reader and the Collector agent respectively, using the anonymous interaction.

```
1  Description description = loadDescription(WSDLFile);
2  Request request = description.getRequest();
3  handler(request, "provideTemp");
4  register(description);
5
6  public void provideTemp(Message m) {
7      Response response = description.getResponse();
8      response.setParamater(``myLocation'', "laboratory");
9      response.setParameter(``myTemperature'', ``32'');
10     tell(m.getSender(), response);
11 }
```

Figure 4.7: Code of the Reader Agent using Anonymous Interaction Mechanism - Java

The code above differs from the one in Figure 4.5 because we add a `register` primitive (line 4). Instead, the code of the new Collector agent, shown below in Figure 4.8, differs from the previous one in the communication primitive used: `askEverybody` instead of `askOne` (line 3). Moreover, lines 9-10 illustrate the use of the `allAnswers` primitive. `allAnswers` is a boolean predicate that returns `true` if the current response is the last reply for the associated `askEverybody`, `false` otherwise.

```
1  description = loadDescription(WSDLFile)
2  request = description.getRequest()
3  self.askEverybody(request, printTemp, fail)
4
5  def printTemp(m):
6      response = description.retrieveResponseFromMessage(m)
7      print 'Temperature in', response.getParameter('myLocation')
8      print 'is:',response.getParameter('myTemperature')
9      if allAnswers():
10         print 'No more data.'
11
12 def fail():
13     print 'No agents found!'
```

Figure 4.8: Code of the Collector Agent using Anonymous Interaction Mechanism - Python

Chapter 5

NOWHERE's Components

Every NOWHERE agent is composed of two main components that work together: the Dispatcher and the Facilitator. Being different entities, possibly written in different programming languages, the Dispatcher and the Facilitator communicates exchanging only a specific `Message` data type. In this Chapter we first describe this `Message` data type, together with the similar `Service` data type, and then we present in details the other components.

5.1 Architecture of a NOWHERE agent

From a logical point of view, every agent is composed of two different components: a *Dispatcher* and a *Facilitator*. The Dispatcher is a language dependent stub that can be extended to create Knowledge Level agents while the Facilitator is a Java object that, as the name suggests, provides facilities to the Dispatcher. A very simple illustration of the internal components of three agents is presented in Figure 5.1.

In the Figure, a KL agent extends the specific Dispatcher, written in the same programming language. The resulting agent communicates with its own Facilitator using the TCP protocol. Thus NOWHERE agents can be programmed in every programming language that supports this network protocol. While a NOWHERE agent is logically composed of a Dispatcher together with a Facilitator, these two components can also physically run on different computers. This is especially useful when using devices with power or computational limitations, such as cellular phones or portable computer devices. The Facilitator is a shared component: different instances of the same component

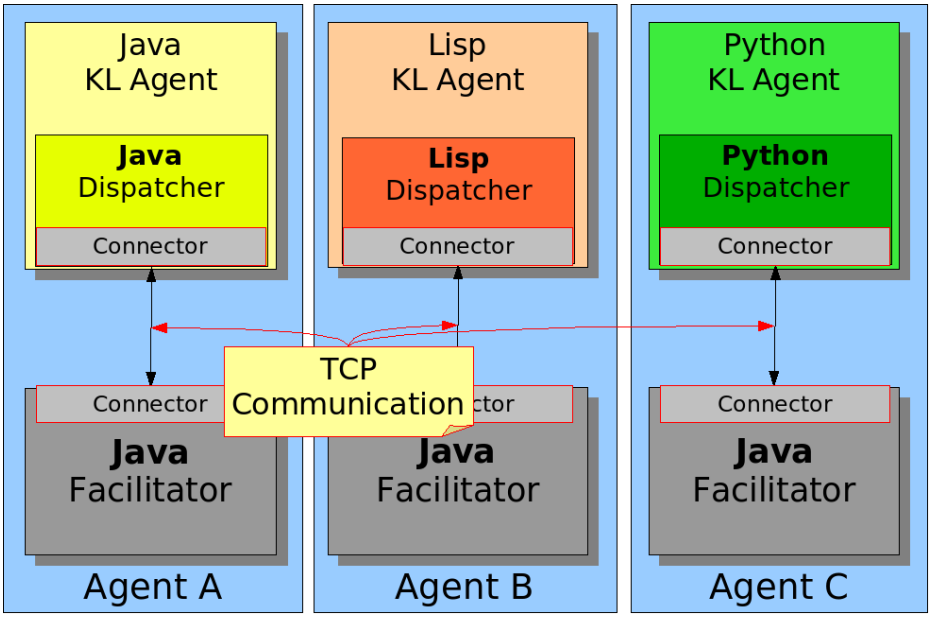

Figure 5.1: Inside an Agent

are used together with different KL agents. Furthermore, a single computer can host several Facilitator instances, providing facilities for a set of agents, for example acting as Facilitator for an entire network.

We paid particular attention to the design of the NOWHERE architecture. The agent Dispatcher is designed to minimise the efforts needed to port it to a new programming language. Moreover, due to the fact that the Facilitator component is shared by the different agent dispatchers, every improvement made in the Facilitator is immediately reflected in all the agents involved in the architecture.

5.2 Messages and Services

The NOWHERE architecture introduces some new data type used to manage messages and services. NOWHERE agents written in different programming languages must be able to understand each others, so that creating standard data types, easy to implement in many languages, is an essential property.

5.2.1 Managing Messages

The NOWHERE architecture allows agents to send and receive a specific data type called `Message`. This data type encapsulates the content of the message in a language-independent way, enabling interoperation among agents written in different programming languages. Every `Message` data type stores information about:

- the `performative` (the type of speech act used, such as `askEverybody` or `askOne`) associated with the message;
- the `sender` and the *receiver* ID;
- the `name` (the topic) of the message;
- the `agentType` and `agentReactiveness` parameters, used for managing the fault tolerant behaviour (for details see Section 6.3);
- other objects that can be associated to the message using the `elements` container.

The `elements` container stores additional data that can be associated to the message. Currently it accepts the following common data types: `String`, `Integer`, `Boolean` and the `Message` data type itself. Furthermore every other data types, like binary objects, can be added as well, converting them as string using the standard Base64 algorithm encoding [48], that is available on almost every programming languages.

Several functions are associated to a `Message` data type in order to manipulate its content. They can be basically divided in two categories: a set of *user functions* that are used by agent developers and a set of *system functions* that are normally used by the NOWHERE architecture itself in order to manage messages. The list of the most important user functions, written in a Java-like form, together with a brief description, can be found in Table 5.1, while the list of the most important system functions, again with a brief comment, can be found in Table 5.2.

While user functions are self-explicative, because they are basically used to get and set elements in a `Message` object, system functions need some clarifications. The `getBase64StringRepresentation` function is used to transform a `Message` structure related to a specific programming language to a language-independent string that can be shared with other agents. The translation is made as follow: first an equivalent string object

`int getAgentReactiveness()`
`setAgentReactiveness(int)`
`int getAgentType()`
`setAgentType(int)`
	Used to get/set the timeout properties of a message

`String getName()`
`setName(String)`
`String getPerformative()`
`setPerformative(String)`
`String getReceiver()`
`setReceiver(String)`
`String getSender()`
`setSender(String)`
	Used to get/set the Name, Performative, Receiver and Sender parameters of a message

`boolean checkMessageName(String)` `boolean checkMessagePerformative(String)`	Used to checking Name and Performative parameters

`void setElement(String, Object)`
`Object getElement(String)`
`Boolean containsElement(String)`
`Iterator<String> getElementsName(String)`
`TreeMap<String, Object> getElements()`
`removeElement(String)`
`setElements(TreeMap<String, Object>)`
`int size()`
	Used to get/set additional data in the message

`Message copyMessage(Message)` `boolean equals(Message)`	Copy and Comparison functions

Table 5.1: `Message` Data Type - User Functions - Java

`String`
`getBase64StringRepresentation(Message)`
`Message getMessageFromBase64String(String)`
	Converts a message to a language-independent one and vice-versa

`String getMessageId()`	Retrieves a fingerprint of the message

Table 5.2: `Message` Data Type - System Functions - Java

is generated, with the full content the message. This temporary object contains every element of the message, together with its data type, processed with an escape function. After that, the string is processed using the standard Base64 algorithm encoding, and the resulting object is returned. The `getMessageFromBase64String` function works exactly in the opposite way, retrieving a `Message` structure, specific for the particular programming language used, from a Base64 encoded string.

The `getMessageId` function is internally used to retrieve a unique ID of a message. To generate the ID, the message is first translated in its Base64 representation using the `getBase64StringRepresentation()` function, and then a "fingerprint" is obtained. While one of the most standard algorithm used for retrieve fingerprinting is still the MD5 [62], we used the "relatively" [1] more secure SHA-1 [3] algorithm. Both the Base64 and the SHA-1 algorithm used in the implementation of the `Message` data type are well known standards and they are already implemented in almost every programming languages.

5.2.2 Managing Services: Description, Request and Response

In the NOWHERE architecture, services differ from messages because they are associated to a description that stores information about the name of the service, the name and data type of its parameters and its return parameters. Messages can then be considered a generalisation of services, because without such description, they can be filled with any parameter.

Services in NOWHERE.

NOWHERE services are quite similar to Web services. They can be imported or exported using the standard Web Service Description Language, WSDL [18]. In NOWHERE, a service description is contained in a `Description` object, that can be retrieved from an existing WSDL file, using the `loadDescription` primitive, or from a specification provided by the user, using the `makeDescription` primitive.

When importing descriptions from WSDL, NOWHERE uses only a subset of the full WSDL specification. In fact, a complete WSDL description consists of six elements:

1. Type, which provides data type definitions used to describe the messages.

[1] In 2005 a Chinese researcher team has found that SHA-1 is not collision-free [72].

2. Message, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.

3. PortType, which is a set of abstract operations. Each operation refers to an input message and output messages.

4. Binding, which specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.

5. Port, which specifies an address for a binding, thus defining a single communication endpoint.

6. Service, which is used to aggregate a set of related ports.

A `Description` object is generated using only the first 3 parameters (Type, Message and PortType), because Binding, Port and Service are used to describe how to physically access the service. Instead, NOWHERE uses its own lower level network to access to its services.

Using the `makeDescription` to create a service is straightforward, and easier than write the associated WSDL code. We illustrate how these primitives work introducing a sample hello service. The hello service has one parameter named `firstParameter`, of type string, and a return parameter named `greetings`, also of type string. An example of this service written in the WSDL, without the information about how to physically access the service, is given in Figure 5.2. An equivalent WSDL code can be exported from a `Description` object created using `makeDescription`, as shown in Figure 5.3.

These two ways of loading a service in the NOWHERE architecture are especially useful in different contexts. When creating a NOWHERE agent, having separated files containing the WSDL description of the services used, facilitates the reuse of the services. In fact, in order to use these services in another application, the developer can retrieve information about the services directly from the WSDL files, without looking into the source code. On the other hand, if we consider agents that create new custom services at runtime, it is easier to create new description directly from the programming language using the `makeDescription` primitive, instead of having to write a correct WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
 targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
 xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <message name="HelloRequest">
  <part name="firstName" type="xsd:string"/>
 </message>
 <message name="HelloResponse">
  <part name="greetings" type="xsd:string"/>
 </message>

 <portType name="Hello_Service">
  <operation name="sayHello">
   <input message="tns:HelloRequest"/>
   <output message="tns:HelloResponse"/>
  </operation>
 </portType>
</definitions>
```

Figure 5.2: The Hello Service - WSDL Description

Description.

This is the main structure that contains information about a specific service. Table 5.3 shows the most important primitives associated to a `Description` object. The `Request` and the `Response` objects are the main entities that can be retrieved from the description of a service. The `getRequest` and `getResponse` functions are used for this purpose, retrieving the information stored in the `Description`. The `retrieveRequestFromMessage` and the `retrieveResponseFromMessage` functions are similar, but retrieve the object from an external message, passed as argument in the functions. This is commonly used while managing requests or responses of services coming from other agents, that have been encoded in a `Message` data type.

```
1  TreeMap<String, String>
2     parameters = new TreeMap<String, String>();
3  parameters.put(``firstName'', ``string'');
4  TreeMap<String, String>
5     returnParameters = new TreeMap<String, String>();
6  returnParameters.put(``greetings'', ``string'');
7  Description desc = makeDescription(
8     ``http://www.ecerami.com/wsdl/HelloService.wsdl'',
9     ``Hello'', parameters, returnParameters);
10 WSDLDescription wsdlD = desc.getWSDL();
```

Figure 5.3: Creating the Hello Service using `makeDescription` - Java

`Request getRequest()` `Response getResponse()`	Retrieves a Request or a Response object from a Description
`String getServiceId()`	Retrieves the unique ID of the service

`Request`
`retrieveRequestFromMessage(Message)`
`Response`
`retrieveResponseFromMessage(Message)`
	Retrieves a Request or a Response object from a Message

Table 5.3: Functions associated to a Description Object - Java

Request and Response.

These objects can be considered templates that can be filled with proper data in order to invoke a service or to provide a response. A `Request` object contains data about the parameters of the service to invoke, while the `Response` object contains data about the parameters of the return value of the same service. These templates can be filled with data using the functions illustrated in Table 5.4.

Services as messages.

While messages contain elements composed of a name and a value, services contain parameters composed of the same properties, a name and a value. Exploiting this similarity, NOWHERE provides a `toMessage` primitive which converts requests or responses into

`Object getParameter(String)`
`void`
`  setParameter(String, Object)`
	Get/Set parameters

`String getServiceId()`	Retrieves the service id
`Message toMessage()`	Converts a `Request` or a `Response` object to a `Message` object

Table 5.4: Functions associated to a Request and a Response Object - Java

messages, so that they can be sent as common `Message` data types. In the same way, requests or responses can be extracted from messages using the appropriate functions provided by a `Description` object.

Comparison with FIPA - JADE JADE implements a Directory Facilitator (DF) agent as specified by FIPA. Agents wishing to advertise their services register them in the DF. Other agents can then search the DF, looking for agents which provide the services they desire.

```
1  DFAgentDescription dfd = new DFAgentDescription();
2  dfd.setName( getAID() )
3  ServiceDescription sd = new ServiceDescription();
4  sd.setType( "buyer" );
5  sd.setName( getLocalName() );
6  dfd.addServices(sd);
7  try {
8     DFService.register(this, dfd );
9  }
10 catch (FIPAException fe) { fe.printStackTrace(); }
```

Figure 5.4: Registering a Service - JADE

The FIPA specification describes several parameters to describe a service:

- name (string) - The name of the service.
- type (string) - The type of the service.

```
1  DFAgentDescription dfd = new DFAgentDescription();
2  ServiceDescription sd = new ServiceDescription();
3  sd.setType( "buyer" );
4  dfd.addServices(sd);
5  DFAgentDescription[] result = DFService.search(this, dfd);
6  System.out.println(result.length + " results" );
7  if (result.length>0)
8     System.out.println(" " + result[0].getName() );
```

Figure 5.5: Searching for a Service - JADE

- protocols (set of strings) - A list of interaction protocols supported by the service.
- ontologies (set of strings) - A list of ontologies supported by the service
- languages (set of strings) - A list of content languages supported by the service.
- ownership (string) - The owner of the service
- properties (set of properties) - A list of properties that discriminate the service.

The main difference between the NOWHERE approach and the FIPA approach, is that FIPA does not allow to search for a service using ACL performative. In the following example (Figure 5.4) the minimal code needed for a JADE agent to register itself as a *buyer* agent is shown. In Figure 5.5 we shown the code needed to search for a specific service. The difference with NOWHERE is that, once discovered all the agents that provide a specific service, the invoking agent must define a way to contact the providing agents, and it must also deal with possible faults.

5.3 Inside the NOWHERE architecture

In this Section we present the two main components of every NOWHERE agent, the Dispatcher and the Facilitator, providing details about their design and showing their internal modules.

5.4 Agent Dispatcher

The agent Dispatcher is a software written in a specific programming language that enables NOWHERE support for that particular language. Thus it can be considered as an extension that enables programs to access the NOWHERE architecture.

Each agent Dispatcher implementation contains a minimal set of structures that provides well defined functionality. The fastest way to develop an agent Dispatcher for a previously unsupported programming language is to replicate these structures, following the code provided for the already implemented dispatchers. The rest of this Section provides a description of the structures that each agent Dispatcher must provide, together with the provided functionality.

5.4.1 The Connector

The Connector is a simple object that provides communication with the local Facilitator. The Connector provides three basic functionality: `read`, `write` and `close`, that act using standard TCP sockets (the `open` functionality is automatically achieved with the creation of this object). The primitives `read` and `write` accept only a `Message` data type, in order to provide the language-independent feature. Three basic steps are followed by the Connector to send a message to the Facilitator:

1. the message is converted into its Base64 representation;
2. the size of the message is then calculated and sent to the Facilitator in a 4 bytes least significant bit form;
3. the Base64 representation of the message is then sent to the Facilitator.

The reading phase works in the opposite way: the first 4 bytes are first read to calculate the total length of the message, and then the message is read and decoded into a local `Message` data type. For simplicity, the `read` function can be written using a blocking primitive, so that the resulting Connector module is developed as a separated thread.

The main code that implements the Connector behaviour in the Dispatcher is a simple fetch/execute cycle, presented in Figure 5.6.

```
1  def run(self):
2      while self.isRunning:
3          messageFromFacilitator = self.read()
4          if messageFromFacilitator == None:
5              # Connection with the Facilitator lost!
6              self.isRunning = False
7          else:
8              # Forward the message to the Dispatcher
9              self.agentDispatcher(messageFromFacilitator)
```

Figure 5.6: Fetch/Execute Cycle - Python

5.4.2 The `dispatcher` Function

Incoming messages are managed using the `dispatcher` function. This function will be automatically called every time an external message is delivered to the agent. The `dispatcher` function is called with the received message as parameter. Figure 5.7 shows the internal state of the agent A that receives three messages in sequence from three different agents B, D and C.

As the messages are received, they are moved in the incoming message queue. Then the Facilitator fetches the first message (the message from agent B) from the queue and forwards it to the Dispatcher for the execution, invoking the `dispatcher` function. While the KL Agent takes the appropriate actions, the Facilitator can accept other incoming messages, moving them into the incoming messages queue. The KL Agent can eventually send some message in response to the one received, and the Facilitator can handle this using only one slot for the outgoing message queue. When the KL Agent terminates the execution of the code associated with the incoming message, the Facilitator will fetch the next message (from agent D) and execute it, as shown in Figure 5.8 .

5.4.3 The Set of ACL

The Dispatcher is the component that provides the entire set of communication primitives to the specific programming language in which it is implemented. Following the idea to minimise the effort while porting the Dispatcher to a new programming language,

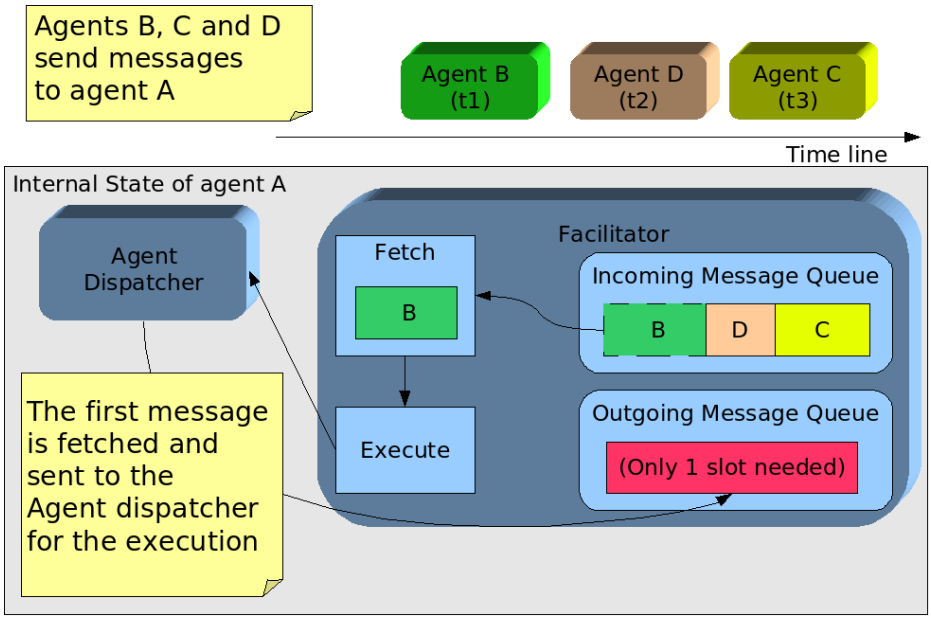

Figure 5.7: Receiving Multiple Messages 1

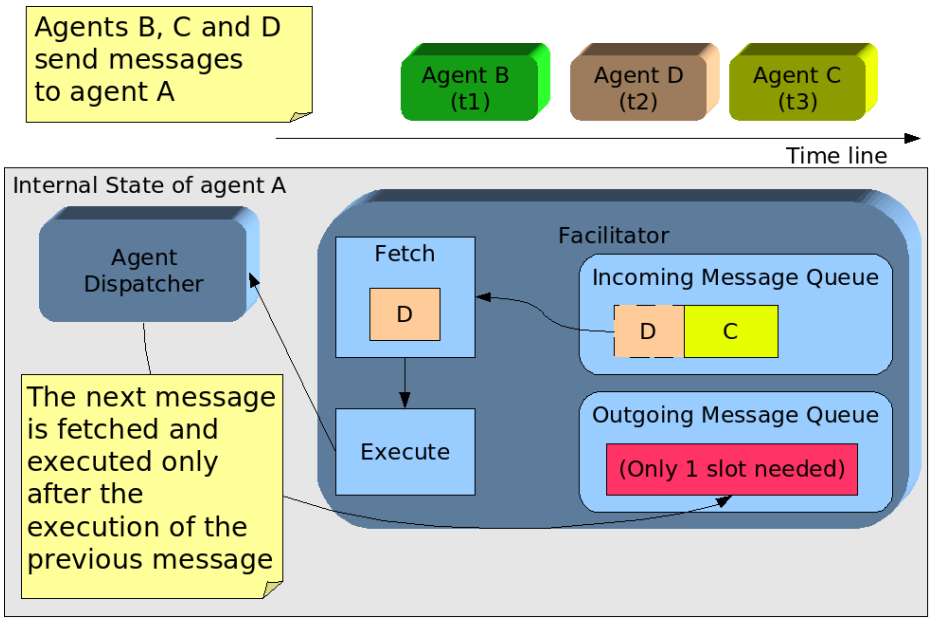

Figure 5.8: Receiving Multiple Messages 2

Affect Dispatcher	Affect Facilitator	Affect both
allAnswers	tell	start
setAgentType	inform	bye
getAgentType		handler
setReactiveness		unhandler
getReactiveness		register
		unregister
		informACK
		askOne
		askEverybody

Table 5.5: ACL Primitives

we moved as much complexity as possible in the Facilitator. In fact many primitives provided by the Dispatcher simply forward a message to the Facilitator, that implements the real behaviour. Following this idea, communication primitives can be divided in three categories, presented in Table 5.5. The categories are:

1. Primitives that affect only the state of the local agent Dispatcher. The execution of such primitives does not imply any communication with the Facilitator.

2. Primitives that affect only the state of the Facilitator. These primitives are directly forwarded to the Facilitator, that will implement the behaviour.

3. Primitives that affect the state of both the Dispatcher and the Facilitator. As consequence of the execution of such primitives, the Dispatcher will modify its state and will also send a message to the Facilitator.

The `allAnswers` primitive can be called by an agent while managing the replies from an `askEverybody` performative. It is used to check if the actual reply is the last one. This primitive is implemented locally in the Dispatcher, thanks to the fact that every reply message of an `askEverybody` primitive reaches the Dispatcher with an extra boolean field that indicates if it is the last one. Also the get/set primitives of the `agentType` and `agentReactive` attribute affect only the Dispatcher. These parameters are automatically attached to every message sent by the agent, so that every message

can have a different timeout property. On the other hand, the `tell` and the `inform` performatives don't affect the local Dispatcher. These primitives are directly forwarded to the Facilitator. The rest of the properties affect both the Dispatcher and the Facilitator. The Dispatcher is affected by the `start` and `bye` primitives (used to start and stop the architecture) because they need to initialise the architecture, and by the other primitives because they all regard services, that are managed locally. The Facilitator is always affected by these primitives because they need to interact with the low level network for initialisation or communication purposes.

5.4.4 The Code Repository

The code repository is a structure that helps to manage both services and continuations in NOWHERE. It provides mechanisms to store and execute functions. Following the idea to move as much complexity as possible into the Facilitator, the Dispatcher job is just to store code in the repository. Then the Facilitator will command the execution (or the removal) of a specific function, reacting to the incoming message. The Table 5.6 lists the set of operations that can be used for a code repository structure.

bindOnAnswer(code, function, args)
bindOnError(code, function, args)
bindService(service, function, args) unbind(service)
isBound(service)
register(service)
runService(service, m, isLastOne)
runErrorService(service)

Table 5.6: Operations supported by the Code Repository Structure

Managing continuation.

Success and failure continuations of a communication primitive like `askOne` or `askEverybody` are stored using `bindOnAnswer` and `bindOnError`, for the success and failure continuation respectively. Using both primitives, the `code` parameter is represented by a string that indicates the unique ID of the specific message, retrieved using

the appropriate command. The `function` parameter represents the associated code to be executed, while the `args` parameter can be used to pass additional values to the code to be executed. An optional failure continuation can be stored using the `bindOnError` primitive, where the parameters have the same meaning of the ones seen before.

Continuations are automatically removed from the code repository when they become useless. Using one-to-one primitives, such as `askOne` or `informACK`, continuations are automatically deactivated just after the execution of the success or the failure continuation. Instead, using the one-to-many primitive `askEverybody`, continuations remain active until the last reply for a given communication primitive.

Managing services.

A service is stored using the `bindService` primitive, that differs from the bind commands seen previously for the first parameter. Using `bindService`, the first parameter is a string that represents the service, usually codified using a URI. The developer can test if a particular service is bound using the boolean predicate `isBind`, and can also unbind the service with the `unbind` command. Binding a service exports an agent capability to other agents, so that the associated Facilitator is able to invoke the service when other agents ask for it using an `askOne` performative. The `register` primitive implements the ACL primitive with the same name. With the registration of a service, other agents are able to invoke it using the anonymous interaction protocol.

Executing services.

The execution of the services provided by the Dispatcher is managed directly by the Facilitator. When an external agent asks for a service, the Facilitator sends an execution message to the Dispatcher. This message is automatically intercepted and executed by the agent Dispatcher, that will act executing the corresponding function stored in the code repository.

5.5 The Facilitator

The Facilitator is a complex distributed component that manages communication for the Knowledge Level agent. Written in the Java programming language, the Facilitator acts

as an interface, connecting the NOWHERE platform to every programming language that provides a Dispatcher.

5.5.1 The Connector

The Facilitator interacts with the Dispatcher using a Connector module, that is almost identical to the Connector module used by the Dispatcher. The Connector provides communication with the Dispatcher, using the TCP protocol, so that the same Facilitator can be used by many Dispatcher implementations.

5.5.2 The Facilitator Core

The Facilitator core is the module that contains the code to route incoming and outgoing messages. From the point of view of the Facilitator, outgoing messages come from the connector module, while incoming messages come from the particular Low Level network plugin chosen. The Facilitator core manages several kinds of messages:

- Internal initialisation messages from Dispatcher to the Dispatcher itself
- Internal messages from Dispatcher to Facilitator
- Outgoing messages from Dispatcher to other agents
- Incoming messages from other agents
- Incoming ask-everybody messages from other agents
- Incoming messages sent by Countdown timers

5.5.3 The Countdown Repository

The countdown repository is a container that stores the countdown objects used in the architecture. A countdown object is a thread associated to an appropriate countdown and to a custom message. The behaviour of the countdown object is to start the countdown timer and just wait until it reaches zero. If the timer reaches zero, than the associated message is sent to the Facilitator, and the appropriate action is taken. If the countdown object is halted before the timer reaches zero, then no action is performed.

In NOWHERE, countdowns are automatically used for two purposes:

- associated to outgoing messages (those that imply a response from other agents), to ensure that the agent will not endlessly wait;
- associated to incoming ask-everybody messages, to ensure that the agent that requested the service will wait until the response is provided.

Timeout handling in NOWHERE is described in Section 6.3.

5.5.4 The Low Level Network Plugin

NOWHERE routes messages to other agents using a low level network plugin. Using different network plugins, NOWHERE has the ability to adapt itself to various scenarios, from real-time systems to networks with a huge number of agents. Network plugins must be implemented as Java classes that extend an abstract class containing a common set of basic primitives. These primitives are described in the abstract class `LowLevelHandler`, and must be implemented by the new plugin. These primitives are presented in Figure 5.9.

```
1  import message.Message;
2
3  public abstract class LowLevelHandler {
4     public abstract void login();
5     public abstract String retrieveId();
6     public abstract void logout();
7     public abstract boolean isConnected();
8     public abstract void sendMessage(Message m);
9     public abstract void join(String group);
10    public abstract void leave(String group);
11    public abstract boolean hasJoined(String group);
12    public abstract void broadcastMessage(Message m);
13 }
```

Figure 5.9: The `LowLevelHandler` Abstract Class

The login process is managed by the primitives `login`, `retrieveId`, `logout` and `isConnected`. While the behaviour of these primitives is self-explicative, it is interesting to note that the ID returned by the `retrieveID` primitive is a string that represents the agent in the specific low level network used. Changing the network plugin will then change the entire set of IDs used (see Section 6.1 for more details about this topic). The primitive `sendMessage` is used to send a message to a specific agent. Every needed parameters, such as the recipient agent, are encoded in the Message data type. The primitives in lines 9-11 are used to manage groups of agents in the network. While IDs are managed by network plugins, groups are managed directly by agents. This is a very important property, because in this way a specific service can be retrieved in the same way using different low level network plugins. Because every group contains agents that provide a particular service, group names are creating using the name of the service associated to them. Finally, the `broadcastMessage` in line 12 is used to send a message to every agent in a specific group.

Chapter 6

Innovative aspects of NOWHERE

In this Chapter we present some interesting details of NOWHERE, comparing them with the JADE platform when possible. We choose JADE for several reasons: because it is one the most used platform, with a big community of both developers and users, because it is a well documented software and finally because - contrary to many other similar abandoned projects - it is still under development.

6.1 Agent Naming

The concept of the agent ID, a unique identifier for each agent, is something usually hardcoded in an agent platform or peer to peer network. The Jabber protocol, for example, defines users in a way similar to email addresses (user@domain/resource), like *juliet@capulet.com/home*. A totally different approach is the one adopted by the JXTA super peer to peer network, where IDs are more similar to the IPv6 numbering, like *urn:jxta:uuid-59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA-53903*.

Due to the fact that NOWHERE supports more than one Network Layer plugin, and that these plugins can be changed at runtime, we decided to manage IDs directly at the Network Layer. Referring to the three layers adopted by NOWHERE (Knowledge Level layer, Architecture Layer and Network layer), only the Network layer will manage directly the agent IDs. The Architecture Layer and the Knowledge Level will gather information about IDs through the Network Layer, so that changing the Network Layer (and then changing the agent naming) will change also the IDs used at Architecture and

Knowledge Level. Using NOWHERE with the JXTA Network Layer, will lead to agents with a JXTA ID. Using NOWHERE with the Jabber Network Layer will lead to agents with a Jabber ID. Following the approach that we have chosen, an agent cannot directly change its ID.

Agent Naming in JADE.

The FIPA agent naming reference model identifies an agent through an extensible collection of parameter-value pairs, called an Agent Identifier (AID). The extensible nature of an AID allows it to be augmented to accommodate other requirements, such as social names, nick names, roles, etc. which can then be attached to services within the AP. An AID comprises:

1. The name parameter, which is a globally unique identifier that can be used as a unique referring expression of the agent. One of the simplest mechanisms is to construct it from the actual name of the agent and its home agent platform address, separated by the @ character.

2. The addresses parameter, which is a list of transport addresses where a message can be delivered.

3. The resolvers parameter, which is a list of name resolution service addresses.

An example of a complete JADE AID is *main@Jeans-Computer.localhost:1099/JADE*. The parameter values of an AID can be edited or modified by an agent, for example, to update the sequence of name resolution servers or transport addresses in an AID.

6.2 Comparing FIPA-ACL Directory Facilitator with NOWHERE's Facilitator

While the term Facilitator is used in both FIPA and NOWHERE implementations, the components identified by this name are very different. In FIPA a Directory Facilitator (DF) is a centralized registry of entries which associate service descriptions to agents IDs. The DF provide primitives to register services and to search for a specific service. Moreover, it allows to search services based on their content rather than their name.

The main difference between this approach and the NOWHERE's approach, is that in FIPA the interaction between the agent and the DF is made outside the ACL. The source code of a JADE agent used to search for a service is shown in Figure 6.1. The agent has the ability to search for services using the `DFService.search` primitive, that is not part of the ACL. After this phase, the requesting agent must interact with the other agents that provide the wanted service, using a primitive of the ACL.

```
DFAgentDescription dfd = new DFAgentDescription();
DFAgentDescription[] result = DFService.search(this, dfd);

sd = new ServiceDescription();
sd.setType( "buyer" );
dfd.addServices(sd);
result = DFService.search(this, dfd);
```

Figure 6.1: Search the DF for a Service - JADE

Generally speaking, using JADE three different phases can be identified:

1. To retrieve the name of the agents that provide the wanted service
2. To request the execution of the service
3. To manage possible failures

Instead, in the NOWHERE architecture, an agent can ask for a service using an ACL primitive. For example, in order to ask for a service S to every agent that provides it, an agent can use the performative `askEverybody`, encoding the service S in the appropriate `Request` object:

```
askEverybody(Request, onAnswer, onFail)
```

This approach represents a very high level communication mechanism, because the agent just asks for a service and manages the replies. The three phases identified for a JADE agent are then achieved with just one primitive.

6.3 Transparent timeouts

NOWHERE uses timeout objects in order to ensure an efficient communication between agents. While the basic usage of timeouts is to avoid agents that endlessly wait for a reply, they can be exploited to provide a framework that can be adaptable to different situations. We call them "transparent timeouts" because they are managed by the architecture itself, so that the user does not have to explicitly deal with them.

In NOWHERE, timeout objects are countdown timers that are activated when a certain primitive is issued or, in some cases, received. Each timeout is associated to a custom message containing an action to do if the countdown timer reaches zero. Usually the action is to execute the failure continuation for the associated primitive.

Every timeout object contains:

- A message, which encodes the action to be taken if the countdown timer reaches zero.
- Two parameters that define the timer: the `agentType` and`agentReactiveness`.

The message associated with every countdown timer is automatically sent using the Facilitator if the countdown reaches zero. The value for the countdown is calculated using the properties `agentType` and `agentReactiveness`.

The `agentType` property can be considered an upper bound of the time that the agent will wait. It defines the maximum time that an agent will wait for external replies. If no replies are given during this time, then the failure continuation is fired.

The `agentReactiveness` is instead the minimum time that an agent will wait for an answer. The Facilitator uses a Java Thread to control each timeout, providing a passive waiting behavior.

6.3.1 Using Timeouts for the `askOne` Primitive

The algorithm implemented by the Facilitator to manage the timeout when using the `askOne` primitive is shown in Figure 6.2.

The algorithm has a loop (lines 3-5) which will end with the success or failure continuation, in lines 4 and 6. The `NeedMoreTime` message is automatically generated and manged by the Facilitator.

1 - The Agent executes the `askOne` primitive.
2 - The associated Facilitator sends the message containing the primitive.
3 - The Facilitator then starts a countdown timer set to the lesser value between `agentReactiveness` and `agentType`.
4 - If, before the countdown reaches zero, the Facilitator receives the reply, than it will halt the countdown and it will forward the received message to the dispatcher (**success continuation**).
5 - If, before the countdown reaches zero, the Facilitator receives a `NeedMoreTime` message, than the `agentType` value is decremented by the actual number of milliseconds already passed since the countdown started.
The algorithm **continues to step 3**.
6 - If the countdown reaches zero, then the message associated to the countdown timer will be forwarded to the Dispatcher (**failure continuation**).

Figure 6.2: Algorithm Used By The Facilitator To Manage Message's Timeout

Timeouts are contained in the `CountdownRepository`, a structure that provide two basic mechanisms: `stop`, to halt a specific timer, and `restart`, to restart it.

In order to explain this algorithm we introduce a simple scenario, in which AgentA executes an `askOne` primitive in order to invoke a service from AgentB. In this scenario there are four possible cases:

1. AgentB replies in due time: the time waited by AgentA for the reply is less than the maximum allowed time set by AgentA (`agentType`). This case is illustrated in Figure 6.3, where FA and FB indicates the Facilitator of AgentA and the Facilitator of AgentB respectively.

2. AgentB has already crashed when AgentA invokes the service. This case is illustrated in Figure 6.4.

3. AgentB receives the request, but it crashes (or a network error occurres) before replying, so that AgentA never receives a proper reply. This case is illustrated in Figure 6.5.

Step	Time	AgentA AgentType: 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_1 < 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
3	$T_2 <$ $T_1 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
4	$T_3 <$ $T_2 + 500$ ms	AgentA executes the success continuation (FA stops its timer)	←	AgentB sends the reply (FB stop its timer)

Figure 6.3: Success Invocation of a Service

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_2 = 500$ ms	AgentA executes the failure continuation (FA stops its timer)		Crash or Network error occurred (If FB not crashed then FB stops its timer)

Figure 6.4: Failure Invocation of a Service (AgentB is already Crashed)

4. AgentB does not reply in due time, that is AgentA does not receive the reply in the maximum allowed time (specified by `agentType`). This case is considered in Figure 6.6.

6.3.2 Using Timeouts for the `askEverybody` Primitive

Timers are also very important for the implementation of the `askEverybody` primitive. In this case the success continuation will be executed by the Facilitator for every given reply. The agent can check if a specific reply is the last one using the `allAnswers` primitive. The algorithm used to manage timers with the `askEverybody` primitive is presented in Figure 6.7.

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB is crashed or not reacheable
2	$T_2 = 500$ ms	AgentA executes the failure continuation (FA stops its timer)		

Figure 6.5: Failure Invocation of a Service (AgentB Crashes before Replying)

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB
1	$T_0 = 0$	AgentA asks AgentB for service S. (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB begin to compute the service. (FB starts its timer: <500ms, up to <4000 ms))
2	$T_1 < 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
3	$T_2 <$ $T_1 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
4	$T_3 <$ $T_2 + 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
...	...	...	...	...
n	T_n=4000 ms	AgentA executes the failure continuation (FA stops its timer)		(FB stop its timer)

Figure 6.6: AgentB does not Reply in Due Time

1 - AgentA executes the `askEverybody` communication primitive.
2 - The associated Facilitator sends the message containing the primitive.
3 - The Facilitator starts a countdown timer set to the lesser value between `agentReactiveness` and `agentType`.
4 - If, before the countdown reach zero, the Facilitator receives a reply, than it will put the message in a temporary queue (with just one free slot).
If the queue already contains a message, than the old message will be forwarded to the Dispatcher (**success continuation**).
5 - If, before the countdown reaches zero, the Facilitator receives a `NeedMoreTime` message, then the `agentType` value is decremented by the actual number of milliseconds already passed since the countdown started.
The algorithm **continues to the step 3**.
6 - If the countdown reaches zero, then if the queue is full, its content will be forwarded to the agent (**success continuation**). In this case the `allAnswers` primitive will return a true value (this is the last reply).
Otherwise, if there is no message in the queue then the message associated to the countdown timer will be forwarded to the Dispatcher (**failure continuation**).

Figure 6.7: Algorithm used by the Facilitator to manage Message's Timeout

The related scenario is AgentA that executes an `askEverybody` primitive to invoke a service from a set of agents (in this case AgentB and AgentC). The case in which AgentB and AgentC replies in due time is illustrated in Figure 6.8.

6.3.3 Timeout values

The `agentType` parameter associates an agent to a specific class of agents with similar interactive characteristics. While any numeric value can be associated to this parameter using the `setAgentType` primitive, NOWHERE specifies a predefined set of values:

- *Real Time Agent*, with maximum waiting time of 2 seconds.
- *Web Agent*, with maximum waiting time of 4 seconds.
- *Worker Agent*, with maximum waiting time of 1 minutes.

Step	Time	AgentA AgentType = 4000 msec Reactiveness: 500msec		AgentB and AgentC
1	$T_0 = 0$	AgentA sends an askEverybody primitive (FA starts its timer: 500 ms up to 4000 ms)	→	AgentB and AgentC begin to compute the service. (FB and FC start their timer: <500ms, up to <4000 ms))
2	$T_1 < 500$ ms	(FA restarts its timer)	←	(FB sends a NeedMoreTime message and restarts its timer)
3	$T_2 < 500$ ms	(FA restarts its timer)	←	(FC sends a NeedMoreTime message and restarts its timer)
4	$T_3 <$ $T_2 + 500$ ms	FA stores the reply in the queue	←	AgentB sends the reply (FB stop its timer)
5	$T_4 <$ $T_3 + 500$ ms	(FA restarts its timer)	←	(FC sends a NeedMoreTime message and restarts its timer)
6	$T_5 <$ $T_4 + 500$ ms	the success continuation is executed (AgentB's reply); FA stores the reply in the queue	←	AgentC sends the reply (FC stop its timer)
7	$T_6 =$ $T_4 + 500$ ms	The last success continuation is executed (AgentC's reply)		

Figure 6.8: Successful Execution of an `askEverybody` Primitive

- *Trusty Agent*, an agent that waits forever until a reply is given.

These values were defined according to the work made by Nielsen in [53], one of the standard reference for the Web usability.

6.3.4 Comparing Timeout Handling with other MASs

Timeouts are widely used in agents platforms. However, the use of timeouts in other architecture lead the developer to deal with low level issues. JADE provides timeouts using behaviors, so that an agent is able to wait for the reply until a specified delay. The source code of an agent that uses this behavior to wait up to 40000 milliseconds for a reply is shown in Figure 6.9. The example comes from the JADE Tutorial [71].

A similar approach can be obtained in KQML, where a specific timeout can be associated to each low level send performative. An excerpt of the source code for the `kqml_send` primitive is presented in Figure 6.10. This primitive supports both blocking and non-blocking behaviors. It contains a pointer to the optional reply, that will be set to `NULL` if a nonblocking behavior is wanted. The return value indicates if the wanted

```
addBehaviour( new myReceiver(this, 40000,
    MessageTemplate.MatchPerformative(ACLMessage.INFORM_REF) {
    public void handle( ACLMessage msg ) {
        if (msg == null)
           System.out.println("Timeout");
        else
           System.out.println("Received:  "+ msg);
    }
});
```

Figure 6.9: JADE Timeout Behavior Example

operation was successful. Again, the developer will have to handle low level issues using this primitive.

```
int kqml_send (int timeout_value, kqml_message *msg,
    kqml_message **reply) {
    /* Sending message...*/
    /* if nonblocking, you are done */
    if (!blocking) {
        if (reply)
           *reply = NULL;
        return 1;
    }
    /* sleep until reply */
    if (reply_message == NULL) {
        return -1;
    }
    if (reply) {
        *reply = reply_message;
    }
    return 1;
}
```

Figure 6.10: KQML Timeout Example

6.4 Adapting NOWHERE to different scenarios using Low Level Network Plugins

The NOWHERE architecture itself cannot be classified as a client-server network or as a peer to peer network, because its behavior is dictated by the Low Level Network used.

For example, when using the JXTA Low Level Network, NOWHERE inherits the features of the JXTA network, such as:

- A truly decentralized platform;
- An infrastructure that supports runtime changes of the network topology, where peers can dynamically change to super peers, providing facilities for other agents (for example to the ones behind firewalls);
- A platform that provides a basic model for security access.

Instead, using the Jabber Low Level Network, NOWHERE inherits other features, such as:

- A decentralized network, with distributed Jabber servers that can interoperate;
- A very stable platform, with tens of thousands of Jabber servers running on the Internet today;
- A fast network, used in many real time application.

Comparison with Jade.

The standard JADE distribution lacks some features that are important in certain applications. However, using third party software it is possible to overcome many limits. The ability to bypass network firewalls, for example, can be achieved using the FIPA Mailbox for Jade [19]. Other JADE extension let the use of a Jabber protocol [39] and a Java Messaging Service [20] for exchanging messages.

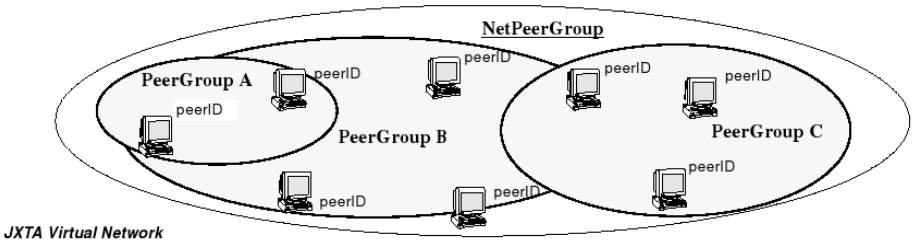

Figure 6.11: JXTA Peergroups. From Project JXTA 2.0 Super-Peer Virtual Network [69]

6.5 Using Groups to reduce the Broadcast Scope

One of the problems that arises using open MASs with many agents is how implement a multicast primitive to a set of agents without have to broadcast the primitive to all agents.

In the NOWHERE architecture, this behavior is achieved using the `askEverybody` primitive. This primitive allow an agent to ask for a service S to every agent in the network that provide this service. In order to avoid to broadcast the message to every agent, causing network congestion, we adopted the same technique used in peer to peer systems, and specifically in the JXTA network: to manage groups of "entities". In the case of JXTA these entities are peers, while in the case of NOWHERE they are agents. In the JXTA peer to peer network, agents can self-organize themselves in peergroups, as shown in Figure 6.11.

Using groups, messages are not broadcasted to every agent in the network, but to every agent in a specific group. A group is a set of agents interested in the same topic, in other words interested in providing or requesting the same service. NOWHERE automatically manages groups of agents, associating them with the services that they provide. Agents that provide the service S (executing the `register` primitive) will be automatically part of a group of agents named S, the same name of the service. Agents that ask anonymously (`askEverybody` primitive) for service S, will temporarily (and automatically) join the group S, broadcasting the message, retrieving the replies and the automatically leave the group. Of course agents can be part of multiple groups at the same time; the Architecture Layer will automatically manage groups for the agent.

6.6 Web Services Integration

Additional mechanisms are needed to enable the full integration of the NOWHERE architecture in contexts with Web services. Agents can always use Web services in the standard way, invoking them directly. However, in our vision agents should use Web services in a simpler way: ideally, the communication from an agent to a Web Server should happen with an ACL primitive, like a communication between two agents. Furthermore, agents should register their competences as Web services, enabling the interaction with conventional programs. NOWHERE provides these features exploiting an external Web Agent Server. A Web Agent Server is an extension of a common Web Server that provides a mechanism for publishing Web services. It is implemented as a Java Servlet hosted in the standard Apache Tomcat [1] servlet container.

6.6.1 Registration of a Web service

NOWHERE provides a functionality which allows Worker agents to register existing Web services locally. As soon as a Web service is registered it becomes reachable as a *virtual agent* and it can be transparently invoked using the $\mathbb{FT}$-$\mathbb{ACL}$ primitives. When an agent requires a competence provided by a virtual agent, its Facilitator recognizes that there is a Web service that matches this competence and maps the ask/tell protocol of the ACL into a standard request/response protocol for Web service invocation. The registration of Web services is a local action: every agent must register the same Web service in order to invoke it with ACL primitives.

6.6.2 Agentification of a Web service

The agentification of a Web service is a more powerful mechanism that integrate Web services in NOWHERE. As shown in Figure 6.12, the Worker agent acts like a wrapper: it provides a competence implemented by the Web service. Every time that an agent asks for that competence, the Worker agent simply invokes the associated Web service and replies with the response gathered from the Web service itself. More than one Web service can be exported by a single Worker agent.

The Worker agent handles the entire communication process with the Web service, converting parameters as needed. The result is that other agents can use the Web ser-

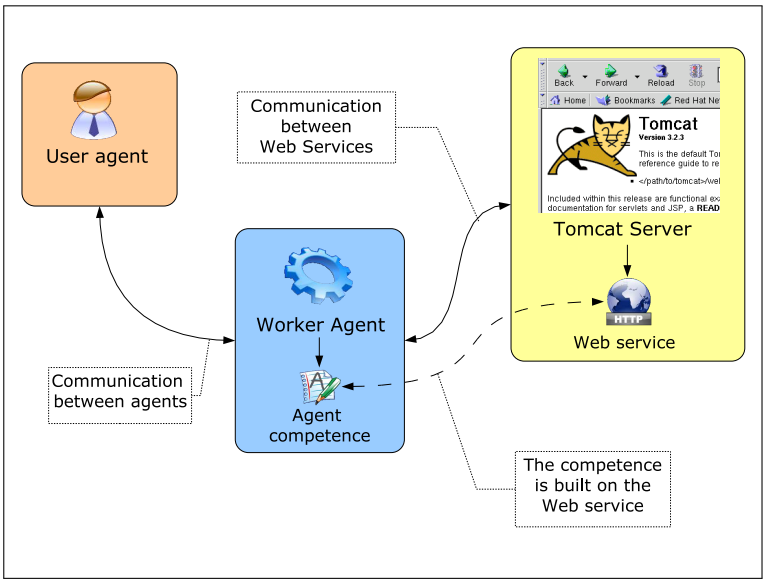

Figure 6.12: Agentification of a Web service

vice with ACL primitives, for example with an `askEverybody` or an `askOne` primitive. Moreover the Worker agent can extend the Web service to provide additional functionality or to integrate it with other Web services.

6.6.3 Exporting Agents as Web services

In a complementary way, a Worker agent could register a particular competence as a Web service, so other programs can interact with the agent without sharing the agent architecture. Every time a Worker agent registers a competence as a Web service, a Web service is generated in the Web Agent Server, with the same name of the registered competence. As shown in Figure 6.13, once the Web service is invoked, the request is translated into an `askOne` performative, then the result is gathered and a standard response is sent back to the caller of the Web service.

Discussion. Both virtual agents and agentified Web services allow agents to discover and to invoke services with an `askEverybody` performative. However, the agentification of a Web service is a more powerful mechanism. For example, agentification can be combined with the export functionality enabling these agents to implement simple Web services or more complex reasoning services. An agentified Web service can be extended

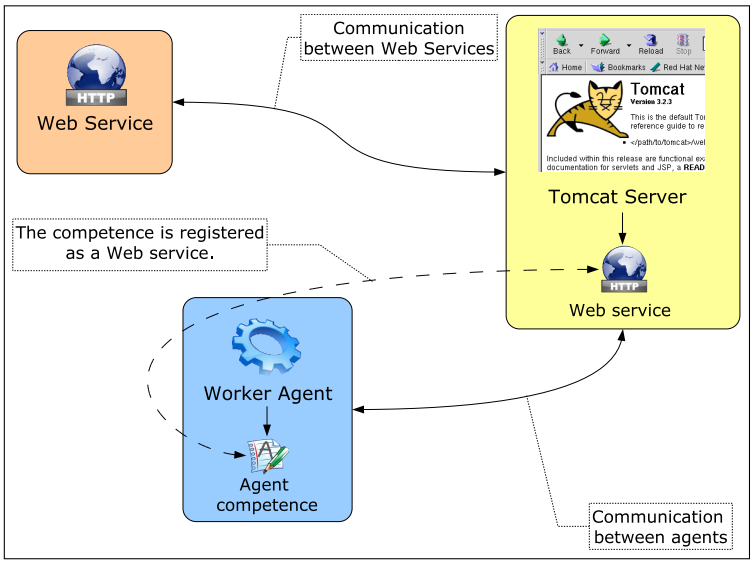

Figure 6.13: Registration of an Agent Competence as a Web service

with an intelligent behavior to provide a more sophisticated version of the same service. This new service can be exported by the agent becoming available to other applications as a standard Web service.

6.6.4 Using NOWHERE in Web service scenarios

Despite our ACL only providing a small set of primitives, they can be successfully exploited in several well known scenarios for Web services usages (as those described in [41] by the W3C Working Group), to come up with original solutions. For example the `tell` primitive is an example of a fire-and-forget to a single receiver scenario, while `askOne` and `tell` can realize a general asynchronous messaging scenario or more complex conversational message exchanges (as the W3C usage scenarios request/response and request with acknowledgment). Another important feature of our ACL is that it supports an *anonymous interaction protocol*. This allows an agent to perform an asynchronous request of services based on contents without knowing the name of the recipient agents (ACL primitive `askEverybody`). If required they can also continue the cooperation using one-to-one communication primitives. In terms of W3C Web services usage scenarios, this is a case of *registry based discovery* where the registry is distributed in all the facilitators. Also the *third party intermediary* W3C usage scenario can be easily realized by means

of our ACL primitives `askEverybody`, `askOne` and `tell`. Moreover, the discovery facility is then integrated with fault-tolerant primitives to manage multiple (non serialized) asynchronous responses.

Web service integration in JADE. There are several approaches for integration Web services in JADE. WS2Jade [52] integrates Web services in the agents architecture, but the integration is only one-way: agents cannot publish their capabilities as Web services.

Another approach is the one proposed by Whitestein Technology, consisting of WSAI and WSIG [21, 38]. WSAI allows Web service clients to use JADE agents services. In order to do this, WSDL files are generated for these agent services. WSIG is a JADE add-on that provides support for bidirectional invocation of Web services from JADE agents, and JADE agent services from Web service clients. It consists of a JADE agent, called the Gateway agent, which controls the gateway from within a JADE container. The implementation of WSIG uses a set of xerces (a XML parser) based codecs to bidirectionally translate ACL messages into WSDL, tModels and SOAP according to the specific context. It also has a socket connection to an Axis [33] Web Server (a Web Service Container) from where agent services can be exposed as callable stubs as if they were Web services. The Axis server also receives calls for invocation from external Web service clients onto agent services. This mechanism is similar to the one adopted by NOWHERE, that provides the same features.

Part III

Case Studies

Chapter 7

Case Studies

In this Chapter we present three case studies that highlight several interesting features of our architecture. In the first case study we present an implementation of the classic Contract Net protocol realized with a state of the art agent platform, and then we provide a detailed comparison with a solution obtained using NOWHERE. The remaining case studies are related the Grid, one of the most interesting application scenario. We provide an evaluation of NOWHERE in terms of usability, reliability and scalability, comparing it, when possible, with other Grid-based solutions.

7.1 The Contract Net Protocol

The purpose of this case study is to compare our architecture with Jade, a state of the art agent platform. We choose the classic Contract Net[65] example because it is an important protocol, fully described in the FIPA specification[6]. The Contract Net protocol allows an agent to distribute tasks among a set of agents by means of negotiation. We only considered a restricted version of the protocol with a single manager agent, the `Initiator`, and a set of worker agents, the `Responders`.

The FIPA specification fully describes a Contract Net protocol version that also includes rejection and confirmation communicative acts. Due to the fact that the Jade software distribution comes with an example of this slightly modified protocol, we used it as subject for our comparison.

In the following we just recall the basic principles of the protocol, described in detail in the FIPA specification. A representation of this protocol is given in Figure 7.1 which

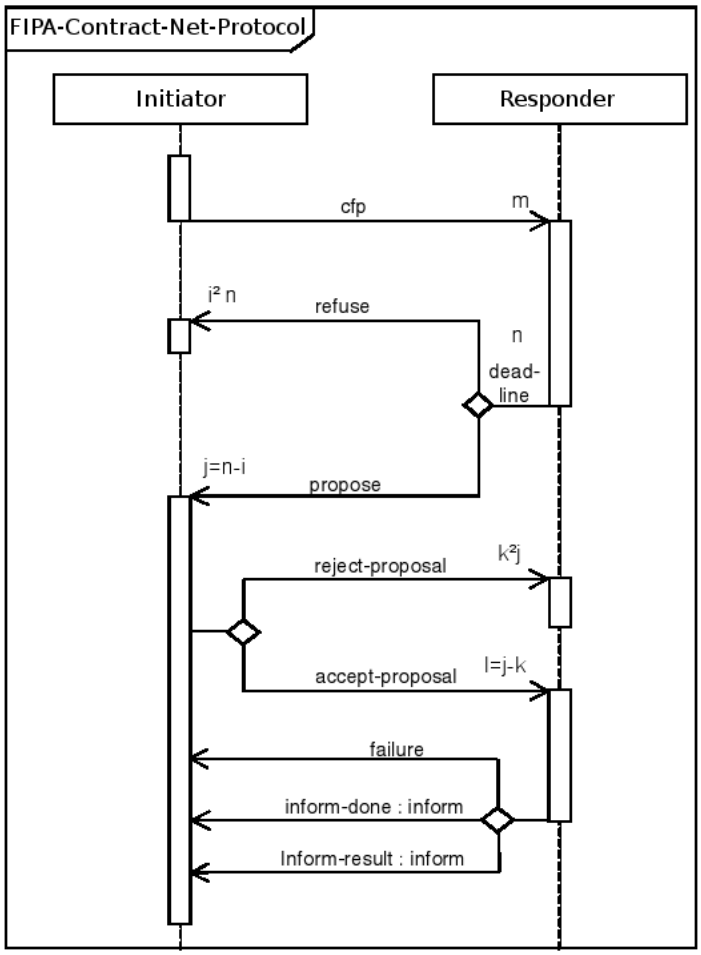

Figure 7.1: FIPA Contract Net Protocol (source: FIPA Specification)

is based on extensions to UML1.x[57]. The sequence diagram describes the inter-agent transactions needed to implement the protocol, where the diamond symbol indicates a decision that can result in zero or more communications being sent, depending on the conditions it contains. According to the FIPA specification, the Initiator agent sends a call for proposal (cfp) act, asking a proposal from every other *m* agents, specifying the task to be done. Responders receiving the call for proposals are viewed as potential contractors and are able to generate *n* responses. Of these, *j* are proposals to perform the task, specified as propose acts. The Responder's proposal includes the preconditions that the Responder is setting out for the task, which may be the price, time when the task will be done, etc. Alternatively, the *i=n-j* Responders may refuse to propose. Once

the deadline passes, the Initiator evaluates the received *j* proposals and selects agents to perform the task; one, several or no agents may be chosen.

Being a FIPA compliant platform, Jade adheres as much as possible to FIPA specifications. In the case of Contract Net, Jade implements the FIPA Contract Net providing facilities that simplify the programming task. In fact, the task of the programmer is just to extend the two Java classes provided for the Initiator and for the Responder. Using the Jade FIPA Contract Net behaviour, written specifically for this protocol, the task of programming these agents is heavily simplified. For example, in order to handle proposals from Responder agents, the developer must only write the proper code inside the `handlePropose` function. The Jade architecture will then invoke this function properly, for each received proposal.

Even if Jade provides ad-hoc facilities to handle this protocol, we think that this can still be an example where programming using the NOWHERE architecture shows some benefit. For the comparison we proceed in this way: first we introduce the algorithm used in the Jade platform (adapted from an example found in the Jade software distribution) and then we provide an equivalent solution for the NOWHERE architecture. We decided to analyze just the Initiator agent, because the Responder agent has a very simple reactive algorithm.

Finally, in this case study we focus on the task of programming the protocol, avoiding technical details such as the initialization of the architecture (both in Jade and in NOWHERE). However, the interested reader can found full source code examples of simple NOWHERE agents in the appendix.

7.1.1 The Initiator agent - Jade

The algorithm implemented by the Initiator agent is composed of 4 main steps:

1. Find the set of available Responder agents;
2. Send a cfp message to Responder agents;
3. Managing replies from Responder agents (proposals, refuses and failures);
4. Evaluate the proposals and accept the best offer;

In the following we describes every single step in details.

1 - Find the set of available agents

The existing source code available in the Jade examples expected the names of the Responders as a list passed as argument to the agent. We modified this code adding the support for the dynamic search for Responder agents, in order to make this case study more adherent to a real world example.

The source code for this first step is presented in Figure 7.2.

```
1  // Fill in a ServiceDescription to find Responders
2  DFAgentDescription df = new DFAgentDescription();
3  ServiceDescription sd = new ServiceDescription();
4  sd.setType("Responder");
5  df.addServices(sd);
6  DFAgentDescription[] agentList = null;
7  try {
8     // Search for other agents
9     agentList = DFService.search(this, df);
10 } catch (Exception e) {
11    e.printStackTrace();
12 }
```

Figure 7.2: Jade Initiator Agent - Find Responders agents

In the Jade platform the task of finding other agents is delegated to the Directory Facilitator component. In order to find other agents, the Initiator should first fill in a Service Description object (lines 1-5). The Service Description object contains information about the resource that we want to find. In this case we used a *type* tag to identify Responder agents. (line 4). The next block of code, lines 6-12, performs a query on the Directory Facilitator and retrieves a list of the available Responder agents.

2 - Send a cfp message to Responder agents

The second step is to send a cfp message to every Responder agent found in the previous step. We used the code shown in figure 7.3.

The code will create a proper cfp message (lines 14-19), specifying every collected agent as receiver, if there are any (line 13). Additionally, the agent sets a maximum

```
13 if (agentList != null && agentList.length > 0) {
14    // Fill the CFP message
15    ACLMessage msg = new ACLMessage(ACLMessage.CFP);
16    for (int i = 0; i < agentList.length; ++i) {
17       msg.addReceiver(((DFAgentDescription)agentList[i]).getName());
18    }
19    msg.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
20    // We want to receive a reply in 10 secs
21    msg.setReplyByDate(new Date(System.currentTimeMillis() + 10000));
22    msg.setContent("dummy-action");
23    addBehaviour(new ContractNetInitiator(this, msg) {
```

Figure 7.3: Jade Initiator Agent - Send cfp message to Responders

timeout of 10 seconds for the proposals (lines 20-21) and the name of the task to be dispatched (line 22). The newly created message is then automatically sent using the `ContractNetInitiator` behaviour (line 23).

3 - Managing replies from Responder

Replies from Responder agents are managed exploiting Jade's FIPA Contract Net behaviour, specifying appropriate code in proper functions. There are three kind of messages to be managed: proposals, refusals and generic failures. Proposals are managed using the `handleProposal` function (source code in Figure 7.4) , refusals are managed using the `handleRefuse` function (source code in Figure 7.5) and failures are managed using the `handleFailure` function (source code in Figure 7.6).

```
24 protected void handlePropose(ACLMessage propose, Vector v) {
25    System.out.println("Agent " +propose.getSender().getName()+
26       " proposed "+propose.getContent());
27 }
```

Figure 7.4: Jade Initiator Agent - Handle proposals

The code used to manage these replies is straightforward, it just prints on the screen the name of the agent that proposed or refused. Two kind of failures are managed: agents

```
28 protected void handleRefuse(ACLMessage refuse) {
29    System.out.println("Agent "+
30       refuse.getSender().getName()+" refused");
31 }
```

Figure 7.5: Jade Initiator Agent - Handle refusals

```
32 protected void handleFailure(ACLMessage failure) {
33    if (failure.getSender().equals(myAgent.getAMS())) {
34       // FAILURE notification from the JADE runtime:  the receiver
35       // does not exist
36       System.out.println("Responder does not exist");
37    }
38    else {
39       System.out.println("Agent "+
40          failure.getSender().getName()+" failed");
41    }
42    // Immediate failure --> no responses from this agent
43 }
```

Figure 7.6: Jade Initiator Agent - Handle failures

that does not exist (those who are not connected anymore in the platform) and generic communication errors.

4 - Evaluate and accept proposals

In this final step the Initiator agent need to evaluate every proposals, selecting the best one for the particular task. This is done using the code in Figures 7.7 and 7.8.

Note that also this code exploits the Jade's FIPA Contract Net behaviour. In fact the function `handleAllResponses` is automatically invoked once every reply is collected or when the selected timeout is expired. The proposal are simply evaluated comparing them against the `bestProposal` variable that stores in every iteration the best proposal received. Replies to the Responder agents are stored in the `acceptances` Java Vector, and are then automatically sent.

```
44 protected void handleAllResponses
45    (Vector responses, Vector acceptances) {
46    // Evaluate proposals.
47    int bestProposal = -1;
48    AID bestProposer = null;
49    ACLMessage accept = null;
50    Enumeration e = responses.elements();
51    while (e.hasMoreElements()) {
52       ACLMessage msg = (ACLMessage) e.nextElement();
53       if (msg.getPerformative() == ACLMessage.PROPOSE) {
54          ACLMessage reply = msg.createReply();
55          reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
56          acceptances.addElement(reply);
57          int proposal = Integer.parseInt(msg.getContent());
58          if (proposal > bestProposal) {
59             bestProposal = proposal;
60             bestProposer = msg.getSender();
61             accept = reply;
62          }
63       }
64    }
```

Figure 7.7: Jade Initiator Agent - Evaluate proposals

Finally, the Initiator agent accepts the best proposal using the code in lines 65-73 of Figure 7.8.

7.1.2 The Initiator Agent - NOWHERE

In the following we provide a similar solution obtained using the NOWHERE architecture. We used only two step for this version.

1 & 2 - Send a cfp message to Responder agents

The code used for sending the cfp is shown in Figure 7.9. However, in this case the

```
65    // Accept the proposal of the best proposer
66    if (accept != null) {
67       System.out.println("Accepting proposal "+
68          bestProposal+" from responder "+bestProposer.getName());
69       accept.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
70    }
71 }
72
73 } // This one closes the addBehaviour statement - line 24
```

Figure 7.8: Jade Initiator Agent - Accepting the best proposal

```
1  // Creating service descriptions
2  TreeMap<String, String> par = new TreeMap<String, String>();
3  par.put("taskName", "string");
4  TreeMap<String, String> returnPar = new TreeMap<String, String>();
5  returnPar.put("proposal", "int");
6  Description cfp = makeDescription("", "cnet-cfp", par, returnPar);
7  par = new TreeMap<String, String>();
8  returnPar = new TreeMap<String, String>();
9  returnPar.put("result", "string");
10 Description action =
11    makeDescription("", "cnet-do-task", par, returnPar);
12 this.setAgentType(10000);
13 // Filling in the cfp message
14 Request r = cfp.getRequest();
15 r.setParameter("taskName", "dummy-action");
16 askEverybody(r, "handleNotify", null, "handleErrors", null);
```

Figure 7.9: NOWHERE Initiator Agent - Sending the cfp

first step is missing. In fact, thanks to the anonymous interaction mechanism, the cfp message can be sent directly to Responder agents, that are automatically discovered. The solution presented in the NOWHERE platform uses two services, one for sending the cfp

```
17 public void handleNotify(Message m) {
18    responses.add(m);
19    if (allAnswer()) {
20       // Evaluate proposals.
21       int bestProposal = -1;
22       String bestProposer = null;
23       Enumeration e = responses.elements();
24       while (e.hasMoreElements()) {
25          m = (Message) e.nextElement();
26          Response r = cfp.retrieveResponseFromMessage(m);
27          int proposal = (Integer) r.getParameter("proposal");
28          if (proposal > bestProposal) {
29             bestProposal = proposal;
30             bestProposer = m.getSender();
31             if (bestProposer != null)
32                inform(bestProposer, new Message("reject-proposal"));
33          }
34          else
35             inform(bestProposer, new Message("reject-proposal"));
36       }
37       // Accept the proposal of the best proposer
38       if (bestProposer != null) {
39          System.out.println("Accepting proposal "+ bestProposal +
40             "from responder "+bestProposer);
41          askOne(bestProposer, action.getRequest(), "taskDone",
42             null, "handleErrors", null);
43       }
44    }
45 }
```

Figure 7.10: NOWHERE Initiator Agent - Managing replies

to Responder agents and one to assign the task to the agent that has replied with the best proposal. The code in lines 1-11 creates the descriptions of these services from scratch,

without relying on external web services definition. The code in line 12 sets the timeout to 10 seconds, accordingly to the Jade's version. A message is then instantiated with proper values (lines 13-15) and sent to the Responder agents using an `askEverybody` primitive (line 16).

3 & 4 - Managing replies Replies from Responder agents are managed using the code shown in Figure 7.10. For every reply received, the Initiator stores the message containing the proposal in a Vector object (line 18). When the last reply is collected, it will execute the code in lines 20-44, that evaluates the proposals and accept the best one. In order to compare the architecture properly, this code is almost identical in both versions. The main difference is that the messages regarding rejected proposals are sent using an `inform` primitive to the less competitive Responder proposals (lines 31-32 and 34-35). The best proposal is finally accepted (lines 38-43).

Failure messages are managed using the code in Figure 7.11, that just prints the errors on the screen.

```
46 public void handleErrors() {
47    System.out.println("Communication error occurred!");
48 }
```

Figure 7.11: NOWHERE Initiator Agent - Handling failures

7.1.3 Discussion

At a first glance, using the Jade's FIPA Contract Net behaviours, the solution obtained are very similar. However, the source code related to the NOWHERE-based solution is smaller than the Jade-based one, even if the Jade version is built using ad-hoc facilities for the Contract Net protocol. This fact highlights the very high level nature of the NOWHERE primitives, even when comparing with a state of the art agent platform.

Analyzing in details the Jade solution, the reader can find that the Jade's FIPA Contract Net behaviour provides mainly two features that are exploited in this example:

- the facility to automate some tasks, like to automatically reply to Responder agents with rejection or acceptance of proposals;

- the facility that allows the developer to consider just the correct proposals, for example implementing only the `handleProposal` function (and then avoiding the implementation of the `handleFailure` function)

However, while these facilities are limited to the FIPA Contract Net scenario, NOWHERE provides high level primitives that may be used in the solution of every kind of problem. The anonymous interaction mechanism, for example, can be used to send a message to every agent in the network that satisfies a set of specific criteria (such as to be a Responder agent).

Moreover, the general idea behind NOWHERE is to simplify the agent programming task, allowing the developer to concentrate in writing the code he is working on, avoiding as much as possible the need to explicitly write code to handle failures. If we focus on the problem that we have to solve (to dispatch a specific task to the agent that replied with the proposal) instead of writing a solution that follows the FIPA Contract Net specification, the resulting source code can be much smaller.

```
17 public synchronized void handleNotify(Message m) {
18    Response r = cfp.retrieveResponseFromMessage(m);
19    int proposal = (Integer) r.getParameter("proposal");
20    if (proposal > bestProposal) {
21       bestProposal = proposal;
22       bestProposer = m.getSender();
23    }
24    if (allAnswer() && bestProposer != null)
25       askOne(bestProposer, action.getRequest(),
26          "taskDone", null,"handleErrors" , null);
27 }
```

Figure 7.12: NOWHERE Initiator Agent - A more compact version

The source code shown in Figure 7.12 represents an alternative solution to manage Responder replies. Using this code, we are able to achieve the same functionality of the Contract Net Initiator agent, but with much less code. The main difference is that here we focus just on the result: we dispatch our task to the agent that replied with the best proposal. This means that we are not required to reply to every Responder (for example

with a proposal rejection), because we are interested just in the best offer.

The provided solutions differ also in the way they manage communication problems. The Jade solution uses a classic timeout approach, whose goal is to wait up to 10 seconds if at least one agent have not replied. Additionally, the NOWHERE platform can be programmed with transparent timeouts (as explained in Section 6.3), which ensure a better exception handling.

7.2 Gridified Connect 4

The purpose of this case study is to show some real application code in order to spot the powerful of NOWHERE in terms of *facility of programming* with respect to common Grid toolkits. The example application is a gridified version of the popular board game Connect 4. We choose this simple example because it is a classic Grid application which has been coded with Globus [46]. In the following we first discuss the case study showing a solution in our architecture and then we compare our solution with the one provided by Globus.

Scenario. The purpose of the game is to align four chips in a chessboard to win, in horizontal, vertical or diagonal way. To choose the next move an intelligent agent works in this way:

- It evaluates the value of each position from the first column to the eighth.
- For each position, it also evaluates the next positions that the adversary could possibly play and reevaluates its tested position accordingly.

Realization. The gridification of this problem is simple: the problem is divided in the eight different evaluations of the current move, one for every possible column choice. The associated function, that calculates the evaluation, is called `simulate`. It evaluates a specific column and can be executed independently by other calls on different columns. The function needs the state of the game and the selected column as parameters and returns the evaluation of the specified move.

In this example we used two kinds of agents: User agents, that interact with the user, and Worker agents, that provide facilities to other agents. User agents are used to manage

the game, the interaction with the player and to display the graphic, while Worker agents are used to evaluate the moves.

In the following we show the programs executed by these agents. We assume the existence of a shared WSDL description for the connect 4 application services. This description contains two services: `getName` and `simulate`. `getName` is a simple service that, once invoked, returns a string containing the name of the agent that provides this service. `simulate` is instead a service with two input parameters, `gameData` and `columnChosen`, that store the game situation and the column chosen by the player respectively. As output, the `simulate` service returns an evaluation of the player's move associated to the specified column.

The first piece of code describes the routine executed by a User agent to gather information about available Worker agents.

```
1  # User agent code to retrieve available Worker
2  # agents - Python version
3  availableAgents = []
4  getNameDesc = loadDescription('connect4NameService.wsdl')
5
6  # Success continuation - store agent name
7  def addAgents(msg):
8      response = getNameDesc.retrieveResponseFromMessage(msg)
9      availableAgents.append(response.getParameter('agentName'))
10
11 # Failure continuation
12 def c4Fail():
13     print('Grid Connect 4 failed:  no agents available!')
14
15 # Retrieve available agents
16 getNameRequest = getNameDesc.getRequest()
17 askEverybody(getNameRequest, addAgents, c4Fail)
```

In this example we used one service, `getName`, shared by the agents. The description of this service is loaded by the `loadDescription` primitive (line 4), while a request template used to invoke the service is retrieved using the `getRequest` primitive (line 16). The gathering process is realized by means of the `askEverybody` primitive (line

17): the User agent makes a request to every other agents, asking for the service used to retrieve agent names. The success continuation is associated to the `addAgents` function that will handle the replies (lines 6-9). The role of this function is to collect the available Worker agents, retrieving their name from the `Response` object. Note that the protocol is deadlock-free also in the worst situation. Indeed, if all the agents that provide the wanted service crashes, then the failure continuation `c4Fail` is called.

In order to provide the service used by the User agent to retrieve names, a Worker agent must register the `getName` service by means of the *register* primitive, as shown in the following code.

```
1  # Worker agent code to register the getName service and
2  # to reply to a query for that service - Python version
3
4  getNameDesc = loadDescription('connect4NameService.wsdl')
5  def sendName(msg):
6      response = getNameDesc.getResponse()
7      response.setParameter('agentName', 'myName')
8      tell(msg.getSender(), response)
9
10 handler(getNameDesc.getRequest(), sendName)
11 register(getNameDesc)
```

Here the function `sendName` is called by the Worker agent to send its own name as a reply to the multicast query previously done by the User agent. The reply is made by means of the `tell` primitive (line 8). When the User agent has received all the replies, it asks the evaluation of the Connect 4 next move to a selection of eight Worker agents. In the following code, the selection criterion is the receiving order, where we suppose that `availableAgents` is a sublist of exactly 8 elements (retrieved from the same variable found in the previous code).

```
1  # User agent code to ask the Connect 4 service and to gather the
2  # results of each agent reply - Python version
3
4  simulateDesc = loadDescription('connect4SimulateService.wsdl')
5  bestChoice = -1
6  associatedColumn = -1
7
8  def gatherResults(msg):
9     response = simulateDesc.retrieveResponseFromMessage(msg)
10    currentEvaluation = response.getParameter('currentEvaluation')
11    column = response.getParameter('column')
12    if (currentEvaluation > bestChoice):
13       bestChoice = currentEvaluation
14       associatedColumn = column
15
16 request = simulateDesc.getRequest()
17 request.setParameter('gameData', gameData)
18 column = 0
19 for agent in availableAgent:
20    request.setParameter('column', column)
21    column = column + 1
22    askOne(agent, request, gatherResults)
```

The User agent simply executes eight times the `askOne` primitive, targeted to specific available Worker agents (previously registered for the Connect 4 service).
The `gatherResults` success continuation specifies the routine that handles the replies.

The last piece of code is the Worker agent algorithm to simulate the move:

```
1  # Worker agent code to simulate the move - Python version
2
3  simulateDesc = loadDescription('connect4SimulateService.wsdl')
4  def simulate(msg)
5      request = simulateDesc.retrieveRequestFromMessage(msg)
6      gameData = request.getParameter('gameData')
7      columnChosen = request.getParameter('columnChosen')
8      evaluation = simulate_AI(gameData, columnChosen)
9      response = simulateDesc.getResponse()
10     response.setParamater('currentEvaluation', evaluation)
11     response.setParameter('column', columnChosen)
12     tell(msg.getSender(), response);
13
14 handler(simulateDesc.getRequest(), simulate)
```

In this code the function `simulate_AI`, (called in line 8) implements the intelligent behavior of the agent. The result of this evaluation, together with the associated column number is returned to the User agent that asked for it (line 12).

Discussion: Comparison With Globus

The main benefit achieved using our architecture instead of other Grid softwares (such as Globus) is usability. In fact, thanks to the expressive power of $\mathbb{FT}$-$\mathbb{ACL}$, the task of writing a concurrent solution to this problem, programming the behavior of Knowledge Level agents, is simplified. For example the code used to retrieve the evaluation of a move done by a Worker agent is:

```
askOne(agent, request, gatherResults)
```

The code of the same example built on the Globus architecture is more complex, because it must explicitly handle the activation of the peers and the ftp connections for sending data. Moreover the syntax of Globus commands is more complex because they deal with many low level issues. On the other hand NOWHERE's primitives have been designed to be at Knowledge Level, thus they only contain requests of the relevant knowledge. To illustrate this point we present below the relative C++ code used in the implementation realized with the Globus toolkit:

```
// Code to ask for a job in Globus
for(i=0;i!=8;i++) {
   cout << "submission on " << node[i] << endl;;
   char tmpc[2];
   sprintf(tmpc,"%d",i);
   // build the RSL commands
   rsl_req = "&(executable=SmallBlueSlave) (arguments=";
   rsl_req += tmpc[0];
   rsl_req += ") (stdout=https://";
   rsl_req +=hostname;
   rsl_req +=":10001";
   rsl_req +=get_current_dir_name();
   rsl_req +="/eval) (stdin=https://";
   rsl_req +=hostname;
   rsl_req +=":10001";
   rsl_req +=get_curren_dir_name();
   rsl_req +="/GAME) (count=1)";
   // submit it to the GRAM
   if (Current.CanPlay(i))
      if (job[i]->Submit(node[i],rsl_req))
   exit(1);
};
```

The above example shows even more clearly the benefits of programming at Knowledge Level.

7.3 Realizing a Distributed Grid Performance System

The role of a Grid Performance System (GPS) is to monitor a laboratory in which workstations are available as computing engines for Grid applications. The goal of the GPS is to make the CPU load of the machines available to the system administrator for monitoring purposes and to Grid applications for distributing tasks appropriately. In [9] a simple scenario for GPS is presented. It describes a laboratory in which workstations are used as desktop systems and are also available as resources for Grid applications. Each

scientist can decide to share its own workstation or not. A Central Server Machine (CSM) on the same local area network is responsible to monitor the CPU load of all the shared workstations and to make the load information available to other systems. A system administrator for the workstations monitors the loads from his/her machine, to ensure that no problems exist. The monitoring machine can be on a different network than the workstations. All the load measurements are archived by an Archiving Service, on a different machine. The archival data could be used for example to analyze daily system load patterns and to identify time periods when the workstations are heavily used.

Scenario. In this case study we illustrate how this scenario can be extended to monitor and integrate in a single Grid performance system machines which belong to different, possibly geographically distributed, local area networks. The aim is to highlight the scalability and reliability features of NOWHERE showing how machines in several subnets can be used for a Grid application despite firewalls and malfunction of nodes. We have tested our GPS with a simple Image Rendering Application.

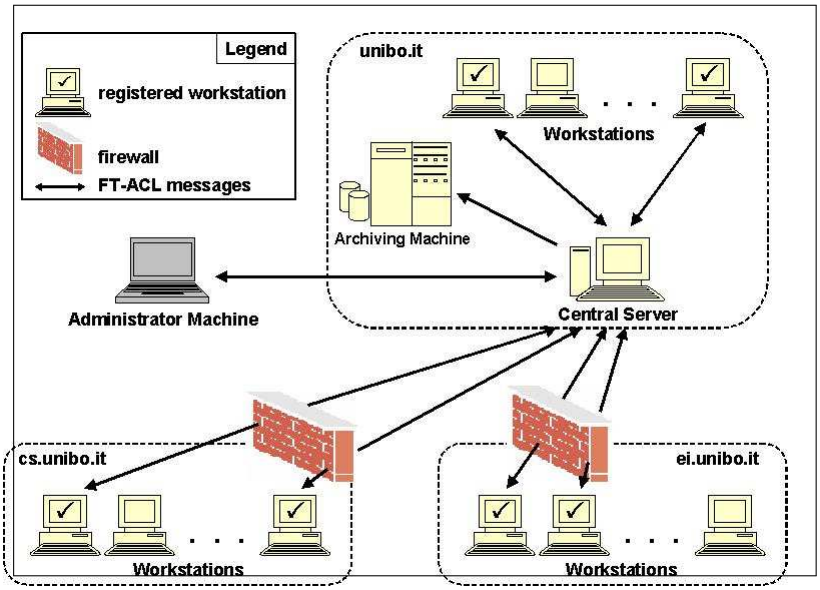

Figure 7.13: Implemented Grid Performance System Scenario

Realization. Consider three laboratories featuring three different ethernet LANs. Figure 7.13 illustrates the components involved in the implemented scenario and their in-

teractions. Note that subnetworks `cs.unibo.it` and `ei.unibo.it` are protected by firewalls.

A Worker agent runs on each workstation, on the Central Server Machine and on the Archiving Machine. A User agent runs on the Administrator Machine to allow the system administrator of monitoring the loads. We illustrate the functionality of our GPS by means of an example.

Image Rendering Architecture

To show how the Grid Performance System could be used, we have built an Image Rendering Architecture on top of it. This scenario can be considered one of the most important in Grid architectures, since it is common to both computer graphic studios and medicine, by the increased availability and diversity of tomographic scanning technologies. In this example we have reduced the complexity of a real world example, rendering only a total of 10 images. Each image needs different time to be rendered: from 30 to 39 seconds each.

To show that machines in different subnets can be used successfully in this application without loosing performance, we first used up to 10 hardware-identically computers, located in a single LAN. Then we repeated the experiment using two different sets of computers from two laboratories in different cities[1].

The implicit client-server architecture works as follows:

- The client just registers the agent as a Grid node providing two different services, one related to the GPS (cpuLoad - that returns both the cpu load and the name of the agent) and one that actually renders the image (renderImage - that gets in input the image to render and returns the rendered image).
- The CSM dynamically retrieves nodes that can be used to render images. If an agent with a low cpu load is found, then the server performs an `askOne` primitives to that agent, specifying the image as argument.

The result of the experiment is shown in the chart in Figure 7.14. The first column represents the time needed by the rendering process itself: at least 345 seconds. As shown

[1]The University of Bologna is organized with a multicampus structure spread across different cities in the Emilia-Romagna region, Italy.

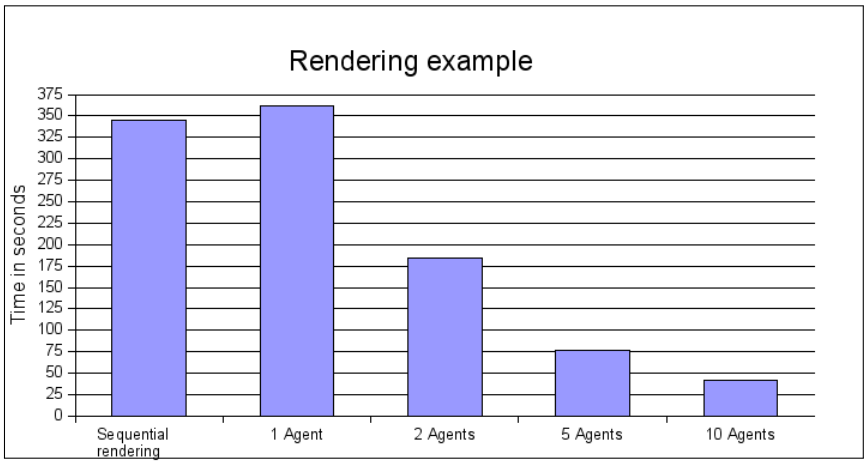

Figure 7.14: Image Rendering Case Study

by the second column, using a grid with only one agent is not, of course, a good idea, due to the time wasted by the server to dynamically search for Grid nodes and to dispatch images to the client. The data in the other columns confirms that the architecture works as expected: with 2 nodes the time needed is exactly one half of the time needed with only one node, and so on. Finally, the last column shows that using 10 nodes to rendering 10 images needs a total of 41 seconds, where the most complex image took 39 seconds to be rendered. The results of the second experiment using computers in two laboratories are identical because the two laboratories are connected by a fast backbone and the time needed to render images is much higher than the time spent in communication.

Discussion

The implemented case study highlights some important features of our architecture:

- Scalability: NOWHERE scales well using computers in different networks, the GPS can be extended to several subnets without considering firewalls and any network restrictions.

- Dynamic platform: the list of workstations available at a given time is dynamically retrieved when the Central Server Machine performs an `askEverybody` primitive. Moreover, at any time unshared workstations can register themselves becoming shared at run-time.

- Fault Tolerance: thanks to the fault-tolerant `askEverybody` primitive, the list of workstations available at a given time does not include crashed workstations. This feature is fundamental in order to avoid infinite waiting and to build real Grid systems.

Chapter 8

Conclusions

The contributions of this Thesis can be summarized in the following main topics.

A Fault Tolerant Agent Communication Language

The proposed ACL is probably the first communication language using high level mechanisms to handle failures. Those mechanisms does not exist in standard ACLs such as FIPA and are based on *continuations* , which can be successful integrated in declarative programming languages in order to realize *Knowledge Level* agents. Moreover, the communication language presented provides multicast primitives that are not available in the current FIPA standard. Another innovation that has been proposed regards transparent timeouts. With transparent timeouts it is possible to send a multicast message to a set of agents, asking for a specific knowledge, without having to deal with agents names or timeout values. Thanks to the high level nature of the communication language, the task of programming agents is heavily simplified, as shown comparing the solutions obtained from a problem solved using our architecture and other state of the art platforms.

A Language-independent Agent Platform

The second important aspect of this work regards the methodology adopted in the realization of the NOWHERE platform. We focused on interoperability, building a modular platform that can be used with different programming languages. The Facilitator component is written in Java, currently one the most used programming language. In this way many low level network protocols, which are available as open source Java libraries, can be integrated. At the same time, however, the Dispatcher component can be

written in programming languages such as Prolog or Lisp, that provide a more rich environment for tasks such as inference or reasoning. Given a specific task, the developer can then choose the best programming language for that specific purpose. Moreover, agents written in different programming languages are able to cooperate. This is a very different approach (agent platforms are usually programmed for a specific language), similar to mechanisms such as OMG's CORBA[2], that integrates distributed objects realized in different programming languages. However our approach extends these mechanisms to support Knowledge Level agents that communicate using high level primitives.

Low Level Network Plugin Layer

During the design of NOWHERE, we defined an interface between the Architecture Level and the Low Level used to send messages. First we have identified a minimum set of requirements needed to build a flexible architecture. Then we have integrated several low level networks, such as Jabber and Jxta. These network protocols provide very different features and can be successfully exploited in different scenarios. Moreover, other network layers can be integrated using a plugin framework, providing support for ad-hoc protocols.

Other innovative aspects regard the bidirectional integration between the ACL and Web services. This mechanism enable the invocation of Web services using the same ACL primitive as well as exporting agent functionality as Web services

8.1 Future Work

We considered many enhancements and new features for our architecture. First of all, we planned to release a first version of NOWHERE as open source software, so that other researchers can test it. We also planned to port the software to other programming languages, to increase interoperability.

Another interesting work will be to investigate the platform with mobile devices, creating useful applications with personal agents features.

Finally, the idea of a *Semantic Web Layer* for NOWHERE is the most interesting research problem of our future work. Both KQML and FIPA messages contain an ontology parameter that can be used to identify a specific vocabulary. According to Gruber [40],

an ontology is a formalization of a shared conceptualization. In Multi-Agent Systems, an ontology can be considered a shared terminology on which different agents must agree, in order to be able to understand each others. In our architecture, agents are not forced to specify an ontology while sending messages. This can be useful when developing small ad-hoc applications, so that the programmer can create agents without having to deal with the construction of an ontology at all.

On the other hand, when developing complex applications it is necessary to have a common terminology that describes the particular domain, to ensure that agents are able to understand each other. To provide such support we plan to extend the core ACL with new primitives and with the related code to handle the Semantic Web extension. The resulting ACL will be available as a "Semantic Agent", that can be extended in order to create KL Semantic Web-enabled agents, replacing the plain Knowledge Level layer.

The overall approach is that, by augmenting Web services with rich formal descriptions of their competence, many aspects of their management (such as Web service discovery, invocation and composition) will become automatic. To realize this vision many open problems need still to be solved. In our opinion, the fundamental ones are:

- Provide a language to semantically express the capabilities of Web services (or service advertisements) and the service requests. Main ongoing works in this direction are OWL-S [58], WSMO [74] or SAWSDL [28].

- Provide an infrastructure which supports the creation of Semantic Web services. The infrastructure must clarify who realize Web services and where the semantic descriptions of Web services are stored (in a centralized or distributed repository). Existing prototypes include WSMF [29] and IRS-III [22].

- Enable automatic discovery and invocation of Web services, that is, enable agents to discover and invoke Web services on the basis of the capabilities that they provide. The discovery problem is also known as "Semantic Matching problem" [59].

Appendix A

Source code examples of NOWHERE agents

In the following we provide the source code of some Java NOWHERE agents. The first example, the Server Agent, is a simple program that just registers two services, ping and bping. The ping service is used to send a ping-like command to a specific agent, in order to check if the agent is alive, while the bping service can be used to broadcast a ping command to a set of agents. The second example shows the code needed to exploit the ping service. Finally, the third source code example shows the anonymous interaction mechanism, exploiting the bping service.

While the core language is still the same in any implementation, some Java-related unique features of NOWHERE are used in the following examples. We provide a brief explanation of these features in the following. Even if the purpose of these examples is just to show the simplicity of programming *real* NOWHERE agents. An agent is created extending the `AgentCL1` class, that provides the Core Language primitives and implementing the `DispatcherInterface` interface, that ensure the creation of the `dispatcher` function. The NOWHERE architecture uses a configuration file to store some parameter, like the name and the password used by the agent. A sample file is generated using the `writeConfigFile` function by the Server program, where the `getFreePort` function is used to detect a free port on a specific host. The `start` function, used to start the architecture, is called specifying a generic message ("init" or "startup"). This message is automatically passed back to the Dispatcher once the network is initialized.

A.1 Server Agent - Java

```
package testAgents;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.net.Socket;
import java.util.Random;
import java.util.TreeMap;

import services.Description;
import services.Request;
import services.Response;

import agent.AgentCL1;
import message.Message;
import agent.DispatcherInterface;

public class Server extends AgentCL1 implements DispatcherInterface {

   String configuration = "";
   Description ping;
   Description bping;

   // Start the agent architecture
   public void runAgentCode(String configDir) {
   // This message m will be automatically sent back after the initalization
      Message m = new Message("init");
      start(configDir, "maya.ei.unibo.it:8080", m);
   }
```

```
// The broadcasted ping service (anonymous interaction mechanism)
public synchronized void bping(Message m) {
    System.out.println("Provide bping service to " + m.getSender());
    Request request = bping.retrieveRequestFromMessage(m);
    System.out.println("Parameter is " + request.
        getParameter("parameter1"));
    Response r = bping.getResponse();
    r.setParameter("returnValue",
        "This is the solution to service BPing");
    tell(m.getSender(), r);
}

// The one-to-one ping service
public synchronized void ping(Message message) {
    System.out.println("Running ping service for " + message.
        getSender());
    Request request = ping.retrieveRequestFromMessage(message);
    System.out.println("Ping parameter is:  " + request.
        getParameter("parameter1"));

    Response response = ping.getResponse();
    response.setParameter("returnValue",
        "This is the solution to service Ping");
    tell(message.getSender(), response);
}

// Every message from other agents is managed here
public void dispatcher(Message m) {
    // Manages the automatic initialization message
    if (m.checkMessageName("init")) {
        System.out.println("Initializing server agent.  " +
            "(ID: " + myId + ")");
        System.out.println("Binding service ping");
        TreeMap<String, String> parameters = new
            TreeMap<String, String>();
        parameters.put("parameter1", "string");
        TreeMap<String, String> returnParameters = new
            TreeMap<String, String>();
        parameters.put("returnValue", "string");
        ping = makeDescription("", "ping", parameters,
            returnParameters);
```

```
            handler(ping.getRequest(), "ping");
            parameters.clear();
            parameters.put("parameter1", "string");
            returnParameters.clear();
            parameters.put("returnValue", "string");
            bping = makeDescription("", "bping", parameters,
                returnParameters);
            System.out.println("Binding service broadcastPing");
            handler(bping.getRequest(), "bping");
            System.out.println("Registering service broadcastPing");
            register(bping);
            System.out.println("Agent initialized.");
        }
        else {
            // Other messages are ignored
            System.out.println("Received message:  " + m.toString());
        }
    }

    // Simple routing to write a standard NOWHERE config file
    public void writeConfigFile(String configFile) {
        // Writing new config file
        try {
            OutputStreamWriter out = new OutputStreamWriter(new
                FileOutputStream(configFile), "US-ASCII");
            String config = "# NOWHERE Configuration file" +
                "\n# Copyright (C) 2004-2006, Applied AI LAB, " +
                "Department of Computer Science, " +
                "University of Bologna" +
                "\n# \n" +
                "\nversion\t\t\t 0.7" +
                "\nfacilitatorHost\t\t localhost" +
                "\nfacilitatorPort\t\t " + getFreePort("localhost") +
                "\nusername\t\t user" +
                "\npassword\t\t password" +
                "\nentryPoint\t\t maya.ei.unibo.it:8080" +
                "\nlocalFacilitator\t true" +
                "\n\n#This is the low level architecture option" +
                "\n#At this time you can use one of 'Jabber' or 'Jxta'" +
                "\nlowLevel\t\t Jabber" +
```

```
            "\n\n# Low Level Jxta-related optional properties" +
            "\nreconfigure\t\t false" +
            "\nserver\t\t\t false" +
            "\n\n";
            out.write(config);
            out.close();
        }
        catch (Exception e) {
            System.out.println("Exception:  " + e);
        }
    }

    // Detects a random free port on the specified host
    private static String getFreePort(String host) {
        // Try to get an open port
        int lowerPort = 49152;
        int higherPort = 65535;
        boolean portIsFree = false;
        while (!portIsFree) {
            int randomPort = new Random().nextInt(higherPort - lowerPort);
            randomPort = randomPort + lowerPort;
            Socket s = null;
            try {
                s = new Socket(host, randomPort);
            }
            catch (IOException ex) {
                portIsFree = true;
                // Returns the discovered free port
                return new Integer(randomPort).toString();
            }
            finally {
                try {
                    if (s != null)
                        s.close();
                }
                catch (IOException ex) { }
            }
        }
        return new Integer(higherPort).toString();
    }
```

```
    // Standard static main method
    public static void main(String args[]) {
        Server ta = new Server();
        String prefix = File.separator + "tmp";
        if (args.length > 0)
            ta.runAgentCode(args[0]);
        else
            ta.runAgentCode(prefix + File.separator + ta.getClass().
                getSimpleName());
    }
}
```

A.2 One to One Communication Example - Java

```
package testAgents;
import java.io.File;
import java.util.Scanner;
import java.util.TreeMap;

import services.Description;
import services.Request;
import services.Response;

import agent.AgentCL1;
import agent.DispatcherInterface;
import message.Message;

public class AskOne extends AgentCL1 implements DispatcherInterface {

    String myId = "";
    Description ping;

    // Initialize the agent architecture
    public void runAgentCode(String configDir) {
        Message m = new Message("startup");
        start(configDir, "maya.ei.unibo.it:8080", m);
    }

    // Handles the initialization message
    public void dispatcher(Message m) {
        if (m.checkMessageName("startup")) {
            TreeMap<String, String> parameters = new
                TreeMap<String, String>();
            parameters.put("parameter1", "string");
            TreeMap<String, String> returnParameters = new
                TreeMap<String, String>();
            parameters.put("returnValue", "string");
            ping = makeDescription("", "ping", parameters,
                returnParameters);
            Request r = ping.getRequest();
            r.setParameter("parameter1", "askOne Parameter");
            System.out.println("Sending ask-one performative.");
```

```
            System.out.print("ID of the agent to ask to:  ");
            // Asks the user for an agent name to ping
            String id = new Scanner (System.in).next();
            setAgentType(WEB_AGENT);
            setAgentReactiveness(WEB_AGENT);
            // Invokes the ping service
            askOne(id, r, "onAnswer", null, "onError", null);
        }
        else {    System.out.println("Received unknown message:  " +
                m.toString());
        }
    }

    // This code handles the reply of the ping service
    public void onAnswer(Message m) {
        System.out.println("Received reply from " + m.getSender());
        Response response = ping.retrieveResponseFromMessage(m);
        System.out.println("Response is " + response.
            getParameter("returnValue"));
        bye();
    }

    // This code will handles problems in the invocation of the' service
    public void onError() {
        System.out.println("Error occurred.");
        System.out.println("Exiting");
        bye();
    }

    public static void main(String args[]) {
        AskOne ta = new AskOne();
        String prefix = File.separator + "tmp";
        if (args.length > 0)
            ta.runAgentCode(args[0]);
        else
            ta.runAgentCode(prefix +File.separator + ta.getClass().
                getSimpleName());
    }
}
```

A.3 Anonymous Interaction Mechanism - Java

```
package testAgents;
import java.io.File;
import java.util.TreeMap;

import services.Description;
import services.Request;
import services.Response;

import agent.AgentCL1;
import agent.DispatcherInterface;
import message.Message;

public class AskEv extends AgentCL1 implements DispatcherInterface {
    String myId = "";
    Description bping;

    public void runAgentCode(String configDir) {
        Message m = new Message("startup");
        start(configDir, "maya.ei.unibo.it:8080", m);
    }


    public void dispatcher(Message m) {
        if (m.checkMessageName("startup")) {
            System.out.println("Sending ask-everybody performative.");
            TreeMap<String, String> parameters = new
                TreeMap<String, String>();
            TreeMap<String, String> returnParameters = new
                TreeMap<String, String>();
            parameters.put("parameter1", "string");
            parameters.put("returnValue", "string");
            bping = makeDescription("", "bping", parameters,
                returnParameters);
            this.setAgentType(100000);
            Request r = bping.getRequest();
            r.setParameter("parameter1", "askEverybody parameter");
            // Request the Bping service to every agent that provides it
            askEverybody(r, "onAnswer", null, "onError", null);
        }
```

```
        else {
            System.out.println("Received unknown message:  " +
                m.toString());
        }
    }

    // Handles every replies for the invoked service
    public void onAnswer(Message m) {
        if (allAnswers())
            System.out.println("Received LAST reply from " +
                m.getSender());
        else
            System.out.println("Received reply from " +
                m.getSender());
        Response r = bping.retrieveResponseFromMessage(m);
        System.out.println("Return value:  " + r.getParameter(
            "returnValue"));
        if (allAnswers())
            // All answers received; exiting
            bye();
    }

    public void onError() {
        System.out.println("Error occurred.");
        System.out.println("Exiting");
        bye();
    }

    public static void main(String args[]) {
        AskEv ta = new AskEv();
        String prefix = File.separator + "tmp";
        if (args.length > 0)
            ta.runAgentCode(args[0]);
        else
            ta.runAgentCode(prefix +File.separator +
                ta.getClass().getSimpleName());
    }
}
```

References

[1] Apache tomcat. Available online at http://tomcat.apache.org/.

[2] The corba website. Available online at http://www.corba.org/.

[3] D. Eastlake 3rd. Us secure hash algorithm 1 (sha1). Available online at: http://www.ietf.org/rfc/rfc3174.txt, 2001.

[4] Inc. Acronymics. Agentbuilder user's guide. Available online at http://www.agentbuilder.com/Documentation/UsersGuide-v1.4.pdf, 2004.

[5] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. Fipa abstract architecture specification. Available online at http://www.fipa.org/specs/fipa00001/SC00001L.html, 2002.

[6] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. Fipa contract net interaction protocol specification. Available online at http://www.fipa.org/specs/fipa00029/SC00029H.pdf, 2002.

[7] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. Fipa agent management specification. Available online at http://www.fipa.org/specs/fipa00023/SC00023K.html, 2004.

[8] J.L. Austin. *How To Do Things With Words*. Harvard University Press, 1962.

[9] R. Aydt, D. Gunter, W. Smith, V. Taylor, and B. Tierney. Simple Case Study of a Grid Performance System. Available online: http://www-didc.lbl.gov/GGF-PERF/GMA-WG/documents.html. Grid Working Draft GWD-Perf-9-3, 2002.

[10] C. Baumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper: a universal agent platform based on omg masif and fipa standards. Available online at: http://cordis.europa.eu/infowin/acts/analysys/products/thematic/agents/ch4/ch4.htm, 2000.

[11] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with jade. In *Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages*, 2000.

[12] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade: a fipa2000 compliant agent development environment. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, New York, NY, USA, 2001. ACM Press.

[13] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[14] F. Brazier, B. Overeinder, M. van Steen, and N. Wijngaards. Generative migration of agents. Proc. of the AISB'02 Symposium on Adaptive Agents and MAS, 2002.

[15] Tryllian Solutions B.V. Public agent runtime environment specifications. Available online at http://www.tryllian.org/docs/are-spec/index.html, 2005.

[16] Knowledge Media Laboratory Corporate Research & Development Center. Beegent - bonding and encapsulation enhancement agent. Available online at http://www2.toshiba.co.jp/rdc/beegent/index.htm.

[17] A. Cheyer and D. Martin. The Open Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4(1):143–148, March 2001. OAA.

[18] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Available online at: http://www.w3.org/TR/wsdl, 2001.

[19] Owen Cliffe. Fipa mailbox for jade. Available online at http://agents.cs.bath.ac.uk/agents/software/fipamailbox/, 2006.

[20] Edward Curry. Mtp implementation based on java messaging service. Available online at `http://ecrg.it.nuigalway.ie/downloads/jmsmtp-latest.zip`.

[21] M. Calisti D. Greenwood. An automatic, bi-directional service integration gateway. In *In Proc. of IEEE Systems, Cybernetics and Man Conference*, 2004.

[22] J. Domingue, L. Cabral, F. Hakimpour, D. Sell, and E. Motta. IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. In *Proceedings of the WIW 2004 Workshop on WSMO Implementations*, Frankfurt, September 29-30 2004.

[23] N. Dragoni and M. Gaspari. An Object Based Algebra for Specifying A Fault Tolerant Software Architecture. *Journal of Logic and Algebraic Programming*, 63(2):271–297, 2005.

[24] N. Dragoni and M. Gaspari. Crash failure detection in asynchronous agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 13(3):355–390, 2006.

[25] N. Dragoni, M. Gaspari, and D. Guidi. An ACL for Specifying Fault-Tolerant Protocols. In *Proceedings of AIIA Conference*, Lecture Notes in Computer Science, pages 237–248, Milan, ITALY, 2005. Springer Verlag.

[26] N. Dragoni, M. Gaspari, and D. Guidi. A fault tolerant agent communication language for supporting web agent interaction. In *Agent Communication: International Workshop on Agent Communication (AC2005), Revised Selected and Invited Papers*. Springer Verlag, 2005.

[27] W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. *In Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, 1996.

[28] Joel Farrell and Holger Lausen. Semantic annotations for wsdl. Available online: `http://www.w3.org/TR/sawsdl/`. W3C Working draft.

[29] D. Fensel, C. Bussler, Y. Ding, and B. Omelayenko. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2), 2002.

[30] Jacques Ferber, Olivier Gutkecht, and Fabien Michel. Madkit development guide. Available online at `http://www.madkit.org/madkit/doc/devguide/devguide.html`.

[31] T. Finin, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In *Software Agents*, pages 291–316. MIT Press, 1997.

[32] FIPA Communicative Act Library Specification. Available online: `http://www.fipa.org/`, 2002. Document number: SC00037J.

[33] The Apache Software Foundation. Axis user's guide. Available online at `http://ws.apache.org/axis/java/user-guide.html`.

[34] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Institute for Intelligent Systems, University of Memphis, Springer-Verlag, 1996.

[35] M. Gaspari. Concurrency and Knowledge-Level Communication in Agent Languages. *Artificial Intelligence*, 105(1-2):1–45, 1998.

[36] M. R. Genesereth and N. J. Nilsson. *Logical Foundation of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1987.

[37] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and state of the art. Technical Report TR2000-365, Dept. of Computer Science, Dartmouth College, 2000.

[38] D. Greenwood and M. Calisti. Engineering web service agent integration. In *In Proc. of IEEE Systems, Cybernetics and Man Conference*, 2004.

[39] Miguel Escriva Gregori, Javier Palanca Camara, and Gustavo Aranda Bada. A jabber-based multi-agent system platform. In *In Proc. of the fifth international joint conference on Autonomous agents and multiagent systems*, 2006.

[40] T. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[41] H. He, H. Haas, and D. Orchard. Web Services Architecture Usage Scenarios. Technical Report NOTE-ws-arch-scenarios-20040211, W3C, February 2004.

[42] A. Hector and V.L. Narasimhan. A new classification scheme for software agents. In *In Proc of ICITA 2005 - Third International Conference on Information Technology and Applications, 2005*, volume 1, pages 191–196, 2005.

[43] Aaron Helsinger and Todd Wright. Cougaar: A robust congurable multi agent platform. In *Proc. of IEEE Aerospace Conference*, 2005.

[44] Intelligent Automation Inc. Cybelepro agent infrastructure user guide. Available online at: `www.opencybele.org/docs/UsersGuideCybeleProVersion1.0.pdf`.

[45] Jack intelligent agents user guide. Available online at: `http://homepage.cs.latrobe.edu.au/mchhabra/doc/Agent_Manual_WEB/index.html`, 2001. Version 3.0.

[46] Bart Jacob, Luis Ferreira, Norbert Bieberstein, Candice Gilzean, Jean-Yves Girard, Roman Strachowski, and Seong (Steve) Yu. Enabling Applications for Grid Computing with Globus. Available online at: `http://www.redbooks.ibm.com/redbooks/pdfs/sg246936.pdf`. IBM Redbooks, SG24-6895, 2003.

[47] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[48] J. Linn. Privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedure. Available online at: `http://www.ietf.org/rfc/rfc1421.txt`, 1993.

[49] Frank G. McCabe and Keith L. Clark. April – agent process interaction language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI volume 890)*, pages 324–340. Springer-Verlag: Heidelberg, Germany, 1995.

[50] S. Mullender. *Distributed Systems*. ADDISON-WESLEY, 1993.

[51] A. Newell. The knowledge level. *Artificial Intelligence*, 19:87–127, 1982.

[52] Thang Xuan Nguyen and Ryszard Kowalczyk. Ws2jade: Integrating web service with jade agents. Technical Report SUTICT-TR2005.03, Swinburne University of Technology, 2005.

[53] J. Nielsen. *Usability Engineering*. MA Academic Press, 1993.

[54] Nortel Networks Corporation, 8200 Dixie Road, Suite 100, Brampton, Ontario, Canada L6R 5P6. *FIPA-OS Developers Guide*, 2001. `http://fipa-os.sourceforge.net/docs/Developers_Guide.pdf`.

[55] H.S. Nwana. Software agents: an overview. *The Knowledge Engineering Review*, 11(3):205–244, 1996.

[56] Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. Zeus: A toolkit for building distributed multi-agent systems. In Oren Etzioni, Jorg P. Muller, and Jeffrey M. Bradshaw, editors, *Third International Conference on Autonomous Agents*, pages 360–361, Seattle, WA, 1999. ACM Press.

[57] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *AOSE*, pages 121–140, 2001.

[58] OWL-S 1.0 Release. Available online: `http://www.daml.org/services/owl-s/1.0/`.

[59] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the first International Semantic Web Conference (ISWC)*, Sardinia, Italy, June 2002.

[60] H. V. D. Paranuk. Applications of distributed artificial intelligence in industry. *Foundations of Distributed Artificial Intelligence*, pages 139–164, 1996.

[61] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. *Jadex: A BDI Reasoning Engine, Chapter of Multi-Agent Programming*. Kluwer Book, 2005.

[62] Ronald Rivest. The md5 message-digest algorithm. Available online at: `http://tools.ietf.org/html/rfc1321`, 1992.

[63] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall Inc, 1995.

[64] J.R. Searle. *Speech Acts*. Cambridge University Press, 1969.

[65] R. G. Smith. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver . *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[66] Simple Object Access Protocol (SOAP). Available online: `http://www.w3.org/TR/soap`.

[67] Nguyen T. and Giang T.Tung. Agent platform evaluation and comparison. Technical Report Pellucid EU 5FP IST-2001-34519 RTD, Institute of Informatics, Slovak Academy of Sciences, 2002.

[68] BBN Technologies. Cougaar architecture document. Available online at `http://cougaar.org/docman/view.php/17/170/CAD_11_4.pdf`, 2004.

[69] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J-C. Hugly, E. Pouyoul, and B. Yeager. Project jxta 2.0 super-peer virtual network. Available online at: `http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf`, 2003.

[70] Venu Vasudevan. Objs technical note: Comparing agent communication languages. Technical report, Object Services and Consulting, Inc., 1998.

[71] Jean Vaucher and Ambroise Ncho. Jade tutorial and primer. Technical report, Dep. d'informatique, Universit de Montrl, 2004.

[72] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. *j-LECT-NOTES-COMP-SCI*, 3621, 2005.

[73] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[74] WSMO Working Group. Web Service Modeling Ontology (WSMO), 2004. WSMO Working Draft D2v1.1.