Dipartimento di Informatica e
Scienze dell'Informazione

Negotiation in Multiagent Systems: Protocols, Ontologies and Applications

by

Daniela Briola

Theses Series **DISI-TH-2011-01**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Negotiation in Multiagent Systems: Protocols, Ontologies and Applications

by

Daniela Briola

August, 2011

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Università degli Studi di Genova
via Dodecaneso 35, I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by

Daniela Briola
DISI, Università degli Studi di Genova, Italy
briola.daniela@gmail.com

Date of submission: February 2011

Title:

Negotiation in Multiagent Systems: Protocols, Ontologies and Applications

Advisors:

Prof. Viviana Mascardi
DISI, Università degli Studi di Genova, Italy
mascardi@disi.unige.it

Prof. Maurizio Martelli
DISI, Università degli Studi di Genova, Italy
martelli@disi.unige.it

Ext. Reviewers:

Prof. Giuseppe Vizzari
Department of Computer Science, University of Milan (Bicocca), Italy
giuseppe.vizzari@disco.unimib.it

Prof. Leon Sterling
Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia
lsterling@swin.edu.au

Abstract

This thesis has its roots in the domain of the "Multi Agent Resource Allocation (MARA)" research field. Its contribution to advancing the research in this field consists in the design of a new and general purpose negotiation protocol, in the definition of the mathematical model upon which it is based, in the implementation of a system called "FYPA (Find Your Path, Agent)", based on this protocol and integrated in a real application of the Ansaldo STS company, and in a detailed comparison of similar protocols, along many different dimensions. The FYPA system is implemented in JADE and also exploits NetLogo to allow users follow the simulation by means of an intuitive graphical interface that we developed. Another system, that we named "MAD MAS" (MAS stands for multiagent system) and that we developed for Ansaldo STS as well, is also described in the thesis. The "MAD MAS" is simpler than FYPA but it shows interesting features since agents are organized in a hierarchical way and have been implemented in a declarative language.

The problem of integrating ontologies into MASs following the FIPA (Foundations of Intelligent Physical Agents) standard is also addressed.

The main research area of the presented work is the one of "resources sharing and their dynamic allocation", that is, we face a set of similar problems where a group of entities dynamically needs to use some resources to achieve one, or more, tasks. In our scope these entities are autonomous agents but the underlying problems are common to very different areas, such as sensors management or autonomous robots coordination.

The first part of the thesis focuses on the general introduction to the research area and to the tools we have used in the thesis.

The second part describes "MAD MAS" and "FYPA", their development, the results of experiments that we carried out and some already existing solutions similar to ours.

The last part presents a detailed survey on how integrating ontologies in MASs and reports our own proposal to achieve this integration.

A Saverio, che ha reso questi tre anni davvero indimenticabili!

Perhaps someday we will discover that space and time
are simpler than the human equation.
(Jean Luc Picard)

Acknowledgements

First of all, I would like to thank Viviana for all her support and help in this thesis and in my research work: she was, and is, irreplaceable! I really not found only an advisor, but above all a colleague and a friend!

And thank to Maurizio: aside from the help with the thesis, he was able to persuade me to start my Ph.D. program when I had many doubts!

Really thank also to Riccardo Caccia, who helped me a lot with the FYPA project!

And I'm really grateful to Gabriele Arecco and Angela Locoro, for the work on the "MAD MAS" and on the Ontology agent, to professor Přemysl Volf who kindly answered me on its MPCA protocol, and to my two reviewers, professor Vizzari and Sterling.

But of course the greatest thanks goes to all those people that in these three years helped, supported and "stood" me: my family and Saverio! I love you!!

And naturally I must thank my friends, because they never stopped telling me "how can YOU design intelligent agents...?", or "now I can understand why trains are always late...": I know you are joking (are you joking, ins't true!?), but you pushed me to do my best, so you are excused!

Thank you all!!

Contents

Chapter 1

Introduction

> "The unexamined life is not worth living"
>
> (Socrate, from Plato's "The Apology of Socrates")

As Socrate already understood about 2400 years ago, the main purpose of human life is the research, the continuous investigation. But of which kind of research are we speaking? What are we searching for? Many philosophers tried to answer this question, as many theologians, scientists and ordinary men, but we cannot know the answer: everyone searches for something different, and sometime we are simply searching "something to look for"...

Anyway, let us consider now only that kind of research that, we hope, can lead to a tangible disclosure, or that deals with something concrete: the scientific research.

As time goes by, science supported humans in their researches more and more, and what was the goal of the research fifty years ago today is only the starting point for a new one. If our forefathers were looking for the "end of the world", now the world is too small for us, and now we are searching for the "end of the universe, or for its beginning...". If our forefathers studied the Nature to understand the human being, now we study the human being to understand if we can create an "artificial human being". Science and philosophy are not so distinguished and many scientific researches today include, for example, social studies, ethic remarks and many other aspects that make the scientific research more complex, but maybe more interesting, than it was in the past.

This thesis is based on what is called "an intelligent software agent": as described later, this research area comes mainly from Artificial Intelligence (AI) and we do not enter here into the old discussion on what AI is because, although it is really interesting, it should require more than one Ph.D. thesis only to summarize this argument and the various opinions on this topic. We limit ourselves to the consideration of the parallelism between an agent and a human, and

we focus on one of the commonly accepted features of a software agent (and of a human): its sociality.

As humans, agents are able to perform a large set of actions, have the ability of learn new operations or facts and can remember their history, have goals and, sometime, also beliefs and intentions. Of course, all these features are limited to their code, to the knowledge representation and so on: we consider only what science allows us to do nowadays, not what maybe we will be able to do in the future!

Moving on with the comparison among humans ad agents, we can assert that agents are also able to interact with other agents. In this case we are considering an "agents society" or, as we usually say, a "Multiagent system (MAS)".

In any society, its members have to interact, and to do this they need some rules, conventions and a common language. As we can argue, a set of agents without norms that regulate their interactions is not a society, is only a chaotic group on entities, where we are no more able to identify anything that seems logical.

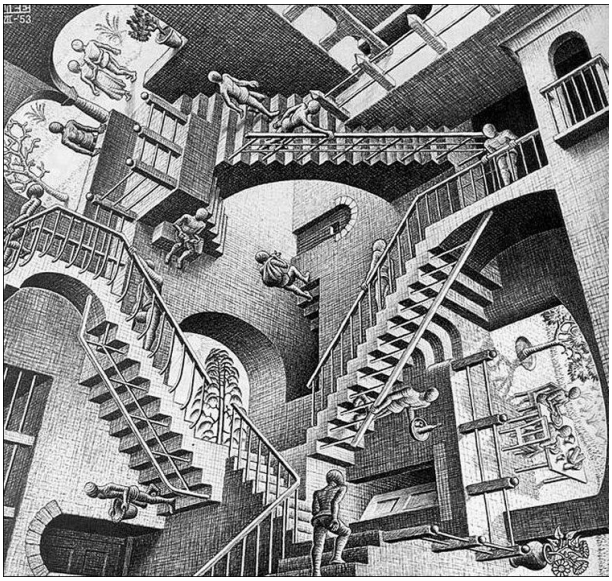


Figure 1: Maurits Cornelis Escher: Relativity (1953)

Note that in a Multiagent system we need these "regulating rules", but they can be explicit or implicit. For example, we know that a human must breath, but this is a "not-written rule", while the fact that "a human must not kill another human" is a rule, a commonly accepted behavior, but nevertheless this convention can be violated. Similarly, in a Multiagent system we can assume that some facts are implicitly valid: if we consider, for example, BDI Agents [108, 103], we know that all the agents will have a representation of what they consider true, but we cannot

assume that they will not lie to each others.

In this thesis we move in the research area of "Multiagent systems" but we limit our research to the "Negotiation in MASs", and partially to the "semantic management in MASs". We are above all interested in which way agents interact: moreover, we consider MASs where agents do not need something like an "institution": a structure similar to a government does not exist, agents autonomously know the rules governing their MAS. The problem is, which rules will they use?

The second area we are interested in is the one of "semantics": humans are able to understand each others because they use a common language. If they have rules, these must be expressed in that language to be understood. This situation should be achieved also in a MAS: how can unknown agents understand each other when they are talking? This is not only a problem of languages, but above all it is a problem of "interpretation". The agents need to chose a syntax (to be used when they communicate) and then must be able to attribute a semantics to it.

The research area dealing with the management of languages and their interpretation in MASs is really broad: we limit ourselves to the study of "Ontologies" and to the possible ways to exploit them in a MAS.

1.1 The collaboration with Ansaldo STS

The work made in this thesis regarding the negotiation in MASs started thanks to a strict collaboration with Ansaldo STS (Genova). The interest of our partner in the Multiagent technology, applied in particular to the real time management of resources allocation, let us find a concrete case study to be used in our academic research. Moreover, we had the rare chance to "face reality" during our work: we used the Ansaldo STS case study as a base for the development of the negotiation protocol that we will describe in the following, and in this way we could also test concretely if the protocol developed using the agent technology was really useful in a real example.

In this thesis we will present the outcomes of two joint works with Ansaldo STS.

The first one is a multiagent system, organized in a hierarchical way, that is able to analyze the logfiles of a set of processes and to discover if these processes have execution problems. When these problems are identified, a solution to solve them is automatically adopted by the MAS or, if the problem cannot be solved in this way, its existence is quickly reported to the assistance center to let the human experts find a solution.

The second result of this collaboration is a multiagent system able to dynamically assign a set of resources (namely, a set of railways in a station) to the trains that need them. Ansaldo STS needed this system to manage the delays of trains entering in a station or to cope with the physical failures of some parts of the station. This system is now used by Ansaldo STS to suggest to

the human user a possible solution to the above mentioned problems (helping him during the complex procedure of choosing how/when to move trains inside the station).

To implement these two systems we used Java, tuProlog and JADE and a previously developed tool, DCaseLP, designed and implemented by other members of our research group.

1.2 The work on ontologies

Our research interest has been always devoted also to the semantic issues related to the multiagent research area.

We analyzed how ontologies could be exploited in a MAS, and we designed a specific agent able to manage the ontologies and to offer a large set of services over them. This problem has been faced by many other researchers, so we reviewed all what we were able to find regarding the integration of ontologies in a MAS: then we made a new proposal to make this integration feasible.

The development of our agent able to deal with ontologies has been successively carried out by other members of our research group.

1.3 Goals and structure of the thesis

> *The primary goal of the thesis is the development of a new interaction protocol for solving a concrete resource allocation problem that shows interesting features from both an industrial and an academic point of view. To this aim, we generalize a real problem proposed by Ansaldo STS, we provide a sound mathematical model for the generalized problem, we design a MAS where agents negotiate in order to solve the generalized problem, and finally we implement and test the designed MAS. Providing an insight of how ontologies can be exploited in systems consisting of negotiating agents is the secondary goal of the thesis.*

The thesis is logically divided into three parts:

- Introduction to the research area and overview of the used tools and techniques
- Detailed description of the developed protocols in the research field of "negotiation in MASs", plus a comparison with other existing solutions
- Detailed analysis of the integration of ontologies in MASs and our proposal

Chapter 2 describes the main features of the Multiagent system research field, reports an overview of the organizational paradigms in MASs and introduces the tools we have used in our work.

Chapter 3 presents the first system we developed for Ansaldo STS (the "MAD MAS"). A hierarchical MAS has been designed and developed to supervise a set of software processes running on different personal computers, all connected by a wired network.

Chapter 4 describes the main MARA problem (MultiAgent Resource Allocation problem) that we analyzed to design and develop a negotiation protocol that coordinates a set of agents which need to use a set of limited resources. This case study has been again provided by Ansaldo STS. The developed system for Ansaldo STS, called "Ansaldo STS FYPA" system, is also described in details.

Chapter 5 deals with the design and code changes needed to make the "Ansaldo STS FYPA" a stand-alone software (that is, independent from the Ansaldo STS external application): it reports the details of the new agents developed for this system (called "Stand-alone FYPA" system) and its graphical interface, made with NetLogo.

Chapter 6 reports a set of examples executed with the "Stand-alone FYPA" system, to let the reader better understand the FYPA protocol.

Chapter 7 presents a detailed comparison made considering four protocols that are similar to ours; these protocols are described independently and then a systematic comparison among them is reported.

Chapter 8 is an extensive description of the already exiting proposals to integrate ontologies in MASs. In the last part of this chapter our own proposal to realize this integration is reported.

Chapter 9 summarizes the evaluations of the two systems developed ("MAD MAS" and "Stand-alone FYPA" system) and of our integration of ontologies in MASs and reports some final considerations on the global research described in this thesis, analyzing the possible future expansions.

Chapter 2

Background

This chapter presents an overview of what we think the reader should know before moving on with the reading of this thesis. We do not intend to cover the next arguments in all the aspects and details because it is not the goal of the thesis. For a domain expert, this chapter will be a summary of the well known features of agents and an overview of the tools and softwares that we used in our work. For an entrant, this chapter will provide the information required to understand the next chapters. Moreover, it is important to underline that we chose to describe in particular those models, techniques and tools that are often used in our research area, the agents negotiation one, and that we exploited during the development of this thesis. So this introduction chapter has no pretension to represent a complete guide to the fields of Multiagent Systems design and development.

2.1 Agent and Multiagent systems

2.1.1 What is an agent? and a MAS?

As Jennings, Sycara and Wooldridge theorize in their well-known article [73] in 1998, the interest in autonomous agents did not emerge from a vacuum. Researchers and developers from many different disciplines have been talking about closely related issues for long time. The main research areas that contributed to the formalization of the "intelligent agents theory" are:

- artificial intelligence [105];
- object-oriented programming [23] and concurrent object-based systems [13, 14];
- human-computer interface design [84].

In this thesis, and in our work in general, we adhere to the definition (called "weak definition") of agent given by Jennings, Sycara and Wooldridge, that specifies the main features of what we call "agent":

"An agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives. There are thus three key concepts in our definition: situatedness, autonomy, and flexibility. [...] By flexible, we mean that the system is:

- responsive: agents should perceive their environment and respond in a timely fashion to changes that occur in it;
- pro-active: agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate;
- social: agents should be able to interact, when appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities."

Nevertheless giving a clear and commonly accepted definition of "what an agent is" it is quite difficult, perhaps impossible, because this entity is differently characterized with respect to the area where it appears.

Many studies have been carried on to model agents as "humans", trying to attribute something like "emotions, ideals, goals..." to these software entities: that is, trying to model them using an anthropomorphic approach. Considering this different point of view, the most accepted definition of an agent is the one (called "strong definition") that adds the "Mentalistic notions" (beliefs, desires and intentions), first introduced by Shoham, Rao and Georgeff [108, 103], to the weak definition mentioned above. Other authors, for example Bates [20], go even further and add "Emotional notions" (friendlyness, trust, untrust...) to the weak and mentalistic ones.

Its not mandatory for any agent to embody all these notions: an agent can show only some of them (or others not cited here) sometimes even mixing some features from the weak and the strong definition.

Anyway, considering the two definitions given above, it is natural to characterize an agent as something that needs other similar entities to act/exist: for this reason, agents are usually able to communicate with other agents. We call "Multiagent system (MAS)" a system composed by several agents that cohabit/cooperate/compete in a specified environment.

As for agents, we cannot give "a final definition" of what a MAS is because, again, being this "software architecture" used in several areas, every field will characterize a MAS with different notions. We report here only the most common features denoting a MAS (that are also those that we assume to hold in our research and in this thesis). In a MAS:

- each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized;
- computation is asynchronous.

Traditionally, research into systems composed of multiple agents was carried out under the banner of Distributed Artificial Intelligence (DAI), and has historically been divided into two main fields: Distributed Problem Solving (DPS) and Multi-Agent Systems (MAS). More recently, the term "multi-agent systems" has come to have a more general meaning, and is now used to refer to all types of systems composed of multiple (semi)autonomous components. Actually, MASs can be considered the "new research frontier" for DAI. As ancestors of agents and MASs we can cite the "Actors" of Agha [13] and the Contract Net Protocol [40].

2.1.2 Multi-Agent Organizational Paradigms

The organization of a multi-agent system is the collection of roles, relationships, and authority structures which govern its behavior. All multiagent systems possess some or all of these characteristics and therefore all have some form of organization, although it may be implicit and informal. In this section we report the main organizational paradigms that can be found in MASs, as described in the work by Horling and Lesser [65].

All approaches have different characteristics which may be more suitable for some problems and less suitable for others.

2.1.2.1 Hierarchies

The hierarchy, or hierarchical organization, is perhaps the earliest example of structured, organizational design applied to multiagent system and earlier distributed artificial intelligence architectures.

Agents are conceptually arranged in a treelike structure, as seen in Figure 1, where agents higher in the tree have a more global view than those below them. In its strictest interpretation, interactions do not take place across the tree, but only between connected entities. The data produced by lower-level agents in a hierarchy typically travels upwards to provide a broader view, while control flows downward as the higher level agents provide direction to those below.

The applicability of hierarchical structuring comes from the natural decomposition possible in many different task environments. The hierarchy's efficiency is also derived from this notion of

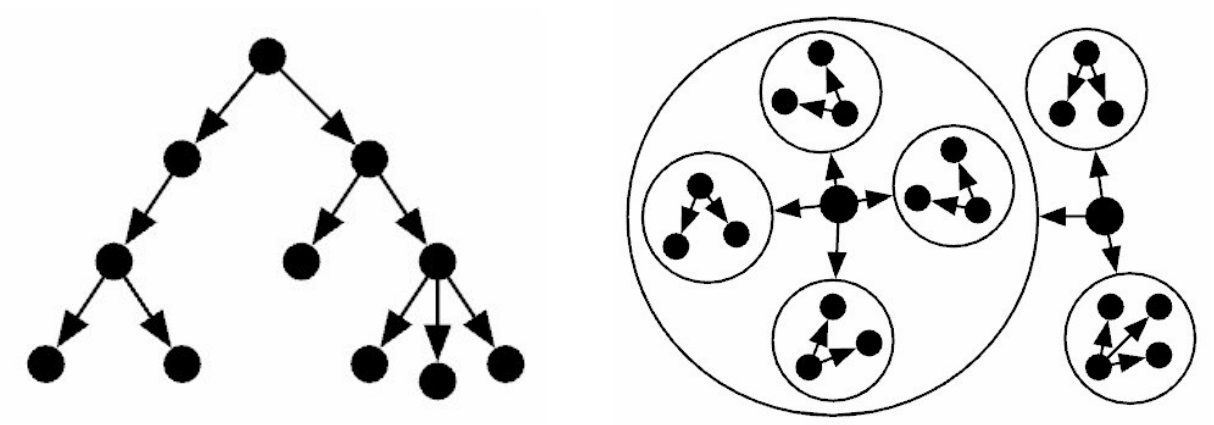


Figure 1: Hierarchical organization and Holarchy

decomposition, because the divide-and-conquer approach it engenders allows the system to use larger groups of agents more efficiently and address larger scale problems.

Using a hierarchy can however lead to an overly rigid or fragile organization, prone to single-point failures with potentially global consequences.

2.1.2.2 Holarchies

The term $holon$ was first coined by Arthur Koestler in his book "The Ghost In The Machine". In this work, Koestler attempts to present a unified, descriptive theory of physical systems based on the nested, self similar organization that many such systems possess. For example, biological, astrological and social systems are all comprised of multi-leveled, grouped hierarchies. Each grouping in these systems has a character derived but distinct from the entities that are members of the group. At the same time, this same group contributes to the properties of one or more groups above it. The structure of each of these groupings is a basic unit of organization that can be seen throughout the system as a whole. Koestler called such units $holons$. An $holarchy$ is a hierarchy of $holons$.

A sample of this organization is shown in Figure 1 (right side). In this diagram, hierarchical relationships are represented as directed edges, while circles represent $holon$ boundaries.

The defining characteristic of a $holarchy$ is the partially-autonomous $holon$. Each $holon$ is composed of one or more subordinate entities, and can be a member of one or more superordinate $holons$.

It would not be incorrect to conclude that a $holarchy$ is just a particular type of hierarchy. If we

relax our definition of hierarchy to allow some amount of cross-tree interactions and local autonomy, the two styles share many of the same features and can be used almost interchangeably.

$Holarchies$ are more easily applied to domains where goals can be recursively decomposed into subtasks that can be assigned to individual $holons$. Given such a decomposition, or a capability map of the population, the benefits the holonic organizations provide are derived primarily from the partially autonomous and encapsulated nature of $holons$. $Holons$ are usually endowed with sufficient autonomy to determine how best to satisfy the requests they receive.

2.1.2.3 Coalitions

The notion of a coalition of individuals has been studied by the game theory community for decades, and has proved to be a useful strategy in both real-world economic scenarios and multi-agent systems. If we view the population of agents A as a set, then each subset of A is a potential coalition. Coalitions in general are goal-directed and short-lived; they are formed with a purpose in mind and dissolve when that need no longer exists, the coalition ceases to suit its designed purpose, or critical mass is lost as agents depart. A population of agents organized into coalitions is shown in Figure 2.

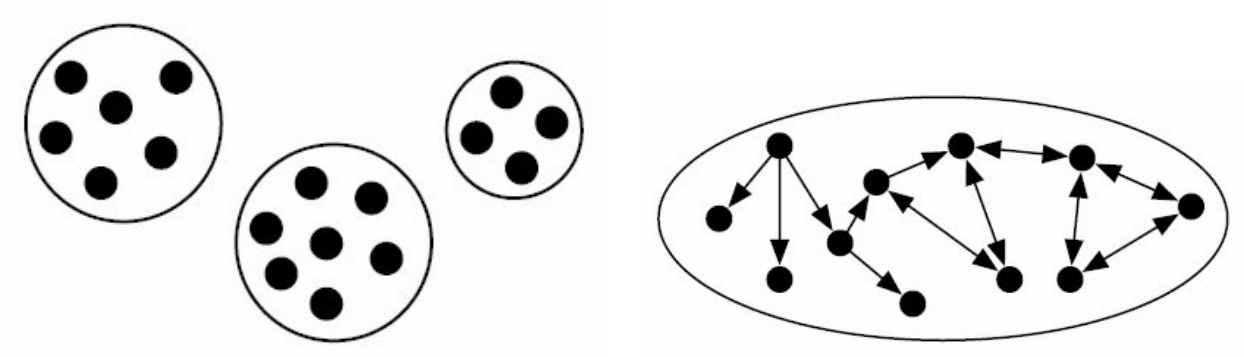


Figure 2: Coalition and Team

Within a coalition, the organizational structure is typically flat, although there may be a distinguished "leading agent" which acts as a representative and intermediary for the group as a whole. Once formed, coalitions may be treated as a single, atomic entity. Therefore, although coalitions have no explicit hierarchical characteristic, it is possible to form such an organization by nesting one group inside another. Overlapping coalitions are also possible. The agents in this group are expected to coordinate their activities in a manner appropriate to the coalition's purpose.

The motivation behind the coalition formation is the notion that the value of at least some of the participants may be greater along some dimension. Analogously, participants' costs may be

lower. This implies that utility can be gained by working in groups. This is the same rationale behind buying clubs, co-ops, unions, public protests and the "safety in numbers" principle. For instance, in an economic domain, a larger group of agents might have increased bargaining strength or other monetary reward.

2.1.2.4 Teams

An agent team consists of a number of cooperative agents which have agreed to work together toward a common goal. In comparison to coalitions, teams attempt to maximize the utility of the team (goal) itself, rather than that of the individual members. Agents are expected to coordinate in some fashion such that their individual actions are consistent with and supportive of the team's goal. Within a team, the type and pattern of interactions can be quite arbitrary, as seen in Figure 2, but in general each agent will take on one or more roles needed to address the subtasks required by the team goal. Those roles may change over time in response to planned or unplanned events, while the high-level goal itself usually remains relatively consistent (although exception handling may promote the execution of previously dormant subtasks).

The primary benefit of teamwork is that by acting in concert, the group of agents can address larger problems than any individual is capable of. Other potential benefits, such as redundancy, the ability to meet global constraints, and economies of scale can also be realized. However, it is the ability of the team (members) to reason explicitly about the ramifications of inter-agent interactions which gives the team the needed flexibility to work in uncertain environments under unforeseen conditions. The drawback to this tighter coupling is increased communication, so the team and joint goal representations, domain characteristics and task requirements are frequently used to determine what level of cooperation (and therefore communication) is needed.

2.1.2.5 Congregations

Similar to coalitions and teams, agent congregations are groups of individuals who have banded together into a typically flat organization in order to derive additional benefits. Unlike these other paradigms, congregations are assumed to be long-lived and are not formed with a single specific goal in mind. Instead, congregations are formed among agents with similar or complementary characteristics to facilitate the process of finding suitable collaborators, as modeled in Figure 3. The different shadings in this Figure represent the potentially heterogeneous purpose behind each grouping, in comparison to the typically more homogeneous coalitions in Figure 2 (left side).

Individual agents do not necessarily have a single or fixed goal, but do have a stable set of capabilities or requirements which motivate the need to congregate. Analogous human structures include clubs, support groups, secretarial pools, academic departments and religious groups, from which the name is derived. Congregating agents are expected to be individually rational,

by maximizing their local long-term utility. Group or global rewards are not used in this formalism. It is this desire to increase local utility which drives congregation selection, because it is the utility that can be provided by a congregation's (potential) members that determine how useful it is to the agent. Agents may come and go dynamically over the existence of the congregation, although clearly there must be a relatively stable number of participants for it to be useful. Agents must also take enough advantage of the congregation so that the time and energy invested in forming and finding the group is outweighed by the benefits derived from it. Since congregations are formed in large part to reduce the complexity of search and limit interactions, communication does not occur between agents in different congregations, although the groups are not necessarily disjoint (i.e., an agent can be a member of multiple congregations). The net result of the congregating behavior is an arrangement that can produce greater average utility per cycle spent computing or communicating.

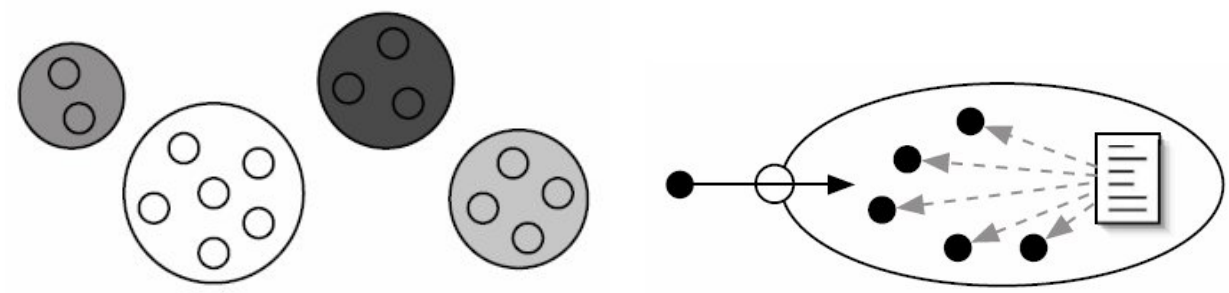


Figure 3: Congregation and Society

Although congregations can theoretically share many of the same benefits of coalitions, their function in current research has been to facilitate the discovery of agent partners by restricting the size of the population that must be searched. The downside to this strategy is that the limited set may be overly restrictive, not contain in some cases the "optimal partner agents". So, in forming the congregation, quality and flexibility can be exchanged for a reduction in time, complexity or cost.

2.1.2.6 Societies

A society of agents intuitively brings to mind a long lived, social construct. Unlike some other organizational paradigms, agent societies are inherently open systems. Agents of different stripes may come and go at will while the society persists, acting as an environment through which the participants meet and interact. A canonical example of this paradigm are the electronic marketplace (discussed in more detail in Section 2.1.2.8), consisting of buyers and sellers striving to maximize their individual utility, or the Electronic Institutions [49, 44, 69].

Agents will have different goals, varied levels of rationality and heterogeneous capabilities; the societal construct provides a common domain through which they can act and communicate. Societies are also more ephemeral constructs than others paradigms we have seen so far. They impose structure and order, but the specific arrangement of interactions can be quite flexible.

Within the society, agents may be sub-organized into other organizations, or be completely unrelated. A second distinguishing characteristic of societies is the set of constraints they impose on the behavior of the agents, commonly known as social laws, norms or conventions. This arrangement is shown abstractly in Figure 3, where the agents within the society have been provided with a set of specified norms. These are rules or guidelines by which agents must act, which provides a level of consistency of behavior and interface intended to facilitate coexistence. For example, it might constrain the type of protocol(s) agents can use to communicate, specify a currency by which they can transfer utility, or limit the behaviors the agent can exhibit in the environment. Penalties or sanctions may also exist to enforce these laws.

In addition to formalizing normative behaviors, mechanisms may also be established to ensure or encourage that such laws are respected. One approach accomplishes this through explicit representations of reputation or trust. Agents's behavior and interactions are observed by its peers and evaluated in the context of the norms it has agreed to. Deviation from those norms will result in a worsening reputation. This decreased reputation can in turn affect the utility the agent obtains, through increased decommitment penalties or competition from more reputable peers. In a rational agent this will serve as a deterrent to violating conventions. A different, but complementary, approach instantiates and enforces social laws using social institutions provided in the environment. Agents are expected to formalize their interactions using contracts, which are independently verified by these institutions, thereby relocating some of the traditionally agent-centric complexity into a service available to the population as a whole.

2.1.2.7 Federations

Agent federations, or federated systems, come in many different varieties. All share the common characteristic of a group of agents which have ceded some amount of autonomy to a single delegate which represents the group. This organizational style is modeled on the governmental system of the same name, where regional provinces retain some amount of local autonomy while operating under a single central government. The delegate is a distinguished agent member of the group, sometimes called a facilitator, mediator or broker. Group members interact only with this agent, which acts as an intermediary between the group and the outside world, as shown in Figure 4. In that Figure each grouping is a federate, and the white agent situated at the edge of each federate is the delegated intermediary.

Typically, the intermediary accepts skill and need descriptions from the local agents, which it uses to match with requests from intermediaries representing other groups. In this way the group is provided with a single, consistent interface. This level of indirection is similar to that seen in

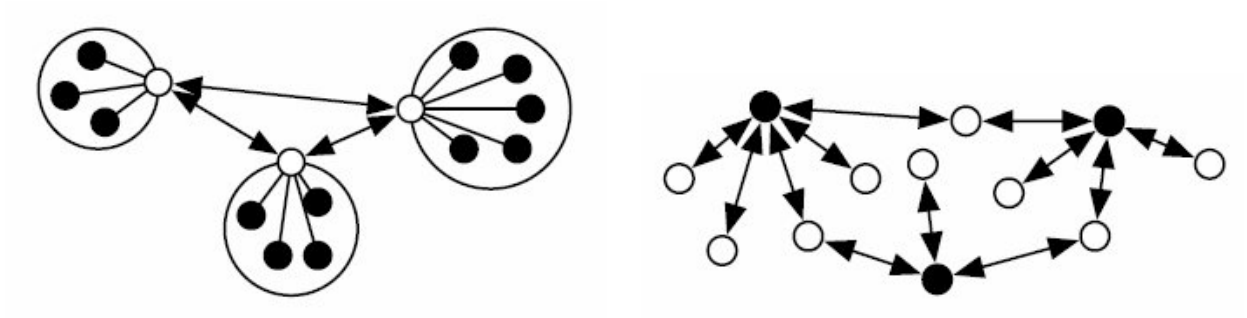


Figure 4: Federation and Market

holons, and provides some of the same benefits.

The intermediary receives messages from its group members. These may include skill descriptions, task requirements, status information and application-level data. These will typically be communicated using some general, declarative communication language which the facilitator understands. Outside of the group, the intermediary sends and receives information with the intermediaries of other groups. This could include task requests, capability notifications and application-level data routed as part of a previously created commitment. Implicit in this arrangement is that, while the intermediary must be able to interact with both its local federation members and with other intermediaries, individual normal agents do not require a common language as they never directly interact. This makes this arrangement particularly useful for integrating legacy or an otherwise heterogeneous group of agents.

The intermediary itself can perform many different tasks. It may act as a translator, perform task allocation, or monitor progress, among other things. An intermediary which accepts task requests and allocates those tasks among its members is known as a broker or a facilitator. As part of the allocation, the broker may decompose the problem into more manageable subtasks. This allows agents to take advantage of all the capabilities of the (potentially changing) federation, without requiring knowledge of which agents perform a task or how they go about doing it. This reduces the complexity and messaging burden of the client, but also has the potential of making the broker itself a bottleneck.

2.1.2.8 Markets

In a market-based organization, or marketplace, as shown in Figure 4, buying agents (shown in white) may request or place bids for a common set of items, such as shared resources, tasks, services or goods. Agents may also supply items to the market to be sold. Sellers (shown in black), or sometimes designated third parties called auctioneers, are responsible for processing bids and

determining the winner. This arrangement creates a producer-consumer system that can closely model and greatly facilitate real-world market economies (Wellman, "Online marketplaces" in [110]). These latter systems fall into the more general category of agent-mediated electronic commerce. Because of this similarity, a wealth of research results from human economics and business can be brought to bear on agent-based markets, creating a solid theoretical and practical foundation for creating such organizations.

Markets are similar to federated systems in that a distinguished individual or group of individuals is responsible for coordinating the activities of a number of other participants. Unlike a federation, market participants are typically competitive. In addition, participants do not cede operational authority to those distinguished individuals, although they do trust the entities managing the market and abide by decisions they make. It is also common for markets to operate as open systems, allowing any agent to take part so long as it respects the system's specified rules and interface. As such, they share some of the benefits and drawbacks of societies.

More generally, Wellman proposes the notion of market-oriented programming [132], which uses the marketplace paradigm as a general programming methodology that can efficiently address resource allocation problems.

Markets excel at the processes of allocation and pricing [133]. If agents bid correctly (i.e., make truthful bids according to their perceived utility gain if they win), the centralized arbitration provided by the auctioneer can result in an effective allocation of goods.

The behaviors embodied in a marketplace, namely the existence of buyers and sellers, a potential multitude of goods, and competition among participants, make such organizations intrinsically linked with the properties of auctions (we do not report a description of the many auction types because this is out of the scope of this thesis).

There are two drawbacks to market-based organizations. The first is the potential complexity required to both reason about the bidding process and determine the auction's outcome.

The second is security; in addition to the practical network-related security issues inherent in any open system, one agent must also be able to verify the validity of the auction approach itself.

As is the case of many open systems, marketplaces are frequently static, pre-existing entities that do not require a formal creation process beyond starting the actual market process (if any) and allowing agents to connect.

2.1.2.9 Matrix Organizations

We have seen that the strict hierarchical organization method is based on a tree-like structure of control. Agents or agent teams report to a single manager, which provides the agents with goals, direction and feedback. Matrix organizations relax the one-agent, one-manager restriction, by permitting many managers or peers to influence the activities of an agent. This forms a mixed-

initiative environment, where successful agents reason about the effects their local actions can have on multiple entities. This is in some sense a closer approximation to how humans exist.

The term matrix organization comes from a grid based view of the participants. One can place managers (black) around a group of $worker$ agents (white), and use a directed edge to indicate authority, as in Figure 5. Alternately, agents are the rows and managers the columns (these sets may overlap), and a check is used to denote where an authority relationship exists. Like the hierarchy's tree, the matrix provides a graphical way to depict which managers can influence the activities of each agent.

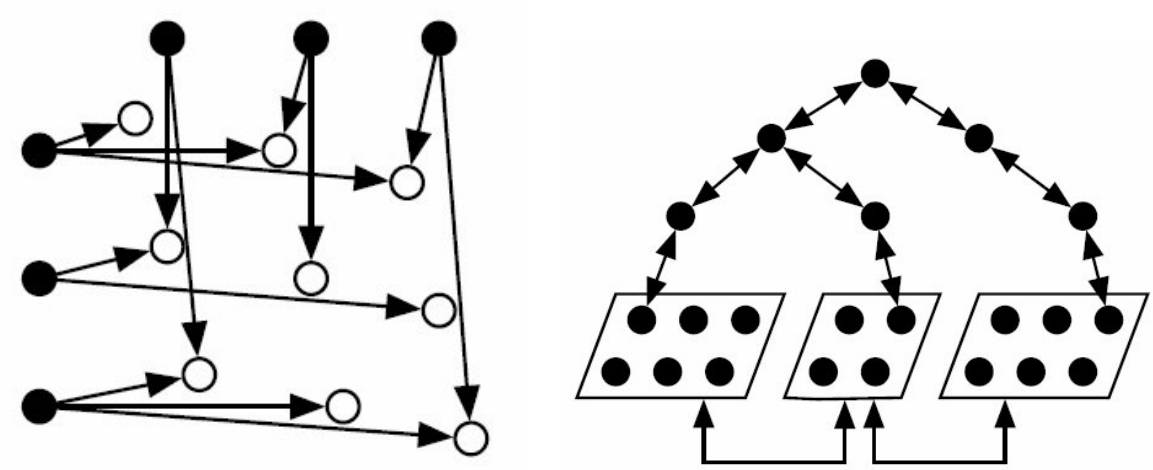


Figure 5: Matrix and Compound organizations

Matrix organizations provide the ability to explicitly specify how the behaviors of an agent or agent group may be influenced by multiple lines of authority. In this way, the agent's capabilities may be shared, and the agent's behaviors (hopefully) influenced so as to benefit all. This is particularly important if the agents themselves are viewed as a functional, limited resources. For example, if a particular skill is needed by two separate tasks, the agent can be used to address both, provided it has sufficient computational power. In the case where the agent has multiple ways of performing a task, it can also choose the method which best satisfies its peers. This sharing come as a price, however, because the shared agent becomes a potential point of contention. If its managers disagree, the agent's actions may become dysfunctional as it is pulled in too many directions at once.

2.1.2.10 Compound Organizations

Not all organizational structures fit neatly into a particular category, and some architectures may include characteristics of several different styles. A system may have one organization for con-

trol, another for data flow, a third for discovery, and so on.

Compound organizations can be overlapped, operating as virtual peers at the same conceptual level, or be nested, so that some subset of agents in a group are organized in a potentially different way within the larger context. A sample such organization is shown in Figure 5, which combines a hierarchy with a set of coalitions. As with singular organizations, they may be created or adapted over time, or they may be instantiated as part of a transient form while a population shifts between organizational styles. Ideally, these compound architectures can use the most effective structure for the particular goal at hand, without limiting options that might be used elsewhere in the system. The tradeoff in this situation is usually one of complexity. Because an individual agent might take on different roles in response to different organizational demands, the agent itself must have sufficient sophistication to act efficiently and asynchronously in all those roles.

The positive and negative characteristics of a compound organization are derived primarily from its constituent parts. However, the interplay between organizations can lead to unexpected consequences. For example, if the distinguished intermediary in a federated system plays a key role in a separate overlay organization, it may be unable to fulfill both roles adequately. Similar to a matrix organization, agents may be faced with conditions where it is not clear which of two competing objectives it should satisfy. Conversely, its knowledge of the requirements of both organizations may enable it to make more globally effective decisions.

2.2 Negotiation in MAS

The literature on Agent-Based Negotiation is huge and scientific research on this topic covers many different aspects, domains and techniques.

Negotiation and the related topics in MASs were, and are, faced by many conferences and workshops. We report a subset of the last ones hold all around the world to underly the interest that these topics raised:

- ACAN 2011: 4th International Workshop on Agent-based Complex Automated Negotiations. Taipei, Taiwan, May 2-6, 2011

  `http://www-itolab.mta.nitech.ac.jp/ACAN2011/`

- ArgMas 2010: 8th International Workshop on Argumentation in Multi-Agent Systems. Taipei, Taiwan, May 3, 2011

  `http://www.mit.edu/~irahwan/argmas/`

- I-MASC 2010: International Workshop on Multi-Agent Systems and Collaborative Technologies. Chicago, Illinois, USA, May 17 - 21, 2010

http://cisedu.us/cis/cts/10/main/storageDocs.jsp?doc=/docs/cts/10/workshops/W04.IMASC.html

- MARA Get-Together: 4th Workshop on Multiagent Resource Allocation. Paris, France, June 17-18, 2010

  http://www.lamsade.dauphine.fr/~maudet/mara/

and other recent conferences and workshops include agent-based and multi-agent negotiation in their topics of interest, for example:

- EUMAS 2010: 8th European Workshop on Multi-Agent Systems. Paris, France, December 16-17, 2010

  http://lipade.math-info.univ-paris5.fr/eumas10/cfp.html

- WI-IAT 2010: 2010 IEEE / WIC / ACM International Conferences. Toronto, Canada, 31 August - 3 September 2010

  http://www.yorku.ca/wiiat10/

- CoopMAS-2011: 2nd Workshop on Cooperative Games in Multiagent Systems. Tapei, Taiwan, May 1-2, 2011

  http://staff.science.uva.nl/~stephane/coopmas11/

- CISIS 2011: 4th Computational Intelligence in Security for Information Systems. Torremolinos, Mlaga, Spain, June 8-10, 2011

  http://gicap.ubu.es/cisis2011/home/home.shtml

Besides conferences and workshops specifically devoted to this research issue, there are hundreds of papers published in journals, books and conferences dealing with agent-based negotiation (also from different perspective). See for example [70, 118, 119, 129, 82, 10], just to cite papers and books written in the last three years.

2.2.1 Trends and research areas

Multiagent research has long been divided into two fields, one concerned with cooperative (benevolent) agents and the other concerned with self-interested agents [46]. There has been very little cross-fertilization of ideas between these fields. Research on self-interested agents is often based on classical game theory with its assumptions of common knowledge among agents and complete rationality of agent reasoning. This is in contrast with the research on cooperative agents which makes no such assumptions; rather, it has generally been based on heuristic

approaches having their roots in knowledge-based AI search, planning and scheduling mechanisms.

However, perhaps the most fundamental and powerful mechanism for managing inter-agent dependencies at run-time is negotiation, that is the process by which a group of agents come to a mutually acceptable agreement on some matter. Negotiation underpins attempts to cooperate and coordinate (both between artificial and human agents) and is required both when the agents are self interested and when they are cooperative.

In this thesis we focus on cooperative agents: in our case study (Chapter 4) we deal with self-interested agents that will anyway negotiate to find a good solution for the system, so our interest is above all on cooperation.

Consequently in this overview chapter we will concentrate on cooperative agents and on the main areas where this kind of agents is required. We will in particular cite the overview on Negotiation in MASs made by Lesser [79], whose research activity, carried out in collaboration with many other scientists, cover a large variety of case studies, domains and applicative projects.

Examples of application domains that have used a multiagent approach for allowing the involved entities to negotiate in a cooperative way are:

- Distributed situation assessment, which emphasizes how (diagnostic) agents with different spheres of awareness and control (network segments) should share their local interpretations to arrive at consistent and comprehensive explanations and responses. Under this category we can find for example:
  1. network diagnosis [114];
  2. information gathering on the Internet [41, 97];
  3. distributed sensor networks [33, 90]).
- Distributed resource scheduling and planning, which emphasizes how (scheduling) agents (associated with each work cell) should coordinate their schedules to avoid and resolve conflicts over resources, and to maximize system output. Under this category we can find for example:
  - factory scheduling [94, 117, 99];
  - network management [11];
  - intelligent environments [22, 68].
- Distributed expert systems, which emphasize how agents share information and negotiate over collective solutions (designs) given their different expertise and solution criteria. Under this category we can find for example:

- concurrent engineering [76];
- network service restoration [38, 71].

The need for a multiagent approach can also come from applications in which agents represent the interests of different organizational entities (e.g., electronic commerce [124] and enterprise integration [19]). Other emerging uses of multiagent systems are in layered systems architectures in which agents at different layers need to coordinate their decisions (e.g., to achieve appropriate configurations of resources and computational processing [130]), and in the design of survivable systems in which agents dynamically reorganize to respond to changes in resource availability, software and hardware malfunction, and intrusions.

In general, multiagent systems provide a framework in which both the inherent distribution of processing and information in an application and the complexities that come from issues of scale can be handled in a natural way.

In these areas the word "Negotiation" is rarely expressly and explicitly used, but anyway in these fields agents usually need to cooperate to achieve a goal, and cooperation is a type of negotiation.

There has also been a long tradition of work dating back to the inception of the field on coordination based on logical reasoning about the beliefs, desires, intentions (BDI) and commitments of agents [39, 102, 109, 62], and more recent work on the use of market mechanisms for solving multiagent resource allocation problems [132].

The synthesis of ideas from each of these different approaches to coordination holds great potential for future developments in the field.

Research works that exploit negotiation for solving Multiagent Resources Allocation problems similar to the one we faced are discussed in Chapter 7.

2.3 Tools and Standards for MASs

Many studies have been carried on the tools to be used in the Multiagent area. We can group these systems in three main sets:

- Tools to design and develop a rapid prototype of a MAS: in this set we can find for example AgentTool [42], The INGENIAS Development Kit (IDK) [60], JACK [67], DCaseLP[89];
- Tools to develop a MAS (as a final application): in this set we can find platforms used to develop complex MASs that are, for example, an expansion of those developed with the previous tools. Some examples are JADE [120], Agent Factory [122], AgentService [2], Zeus [96];

- Tools to develop MAS for "agent-based simulation": in this set we can mention some environments to design and develop simulations where many similar agents interact (simulation where the emerging behavior of the system is the main aspect). Some examples are NetLogo, Mason [59] and SWARM [116].

We cannot analyze all of them here: anyway there are surveys ([47],[58],[16]) and even workshops (`http://itmas2010.gti-ia.dsic.upv.es/Sitio_web/Home.html`, `http://lia.deis.unibo.it/confs/lads/lads007.pdf`) describing the most used tools and comparing them. In this section we just discuss those tools that we used during our Ph.D. Thesis development.

2.3.1 DCaseLP

We start this tools analysis from the area that deals with the design and the development of a rapid prototype of a MAS.

In this section we report the work of Mascardi, Gungui and Martelli, that designed and developed DCaseLP, that we used in our research to exploit tuProlog using JADE.

DCaseLP [89] stands for Distributed Complex Applications Specification Environment based on Logic Programming. Although initially born as a logic based framework, as the acronym itself suggests, DCaseLP has evolved into a multi-language prototyping environment that integrates both imperative (object-oriented) and declarative (rule-based and logic-based) languages, as well as graphical ones. The languages and tools that DCaseLP integrates are UML and an XML-based language for the analysis and design stages, Java, Jess [91] and tuProlog [43] for the implementation stage, and JADE for the execution stage. Software libraries for translating UML class diagrams into code and for integrating Jess and tuProlog agents into the JADE platform are also provided.

2.3.1.1 Developing Stages

DCaseLP provides the languages and tools that support a MAS developer in the engineering stages from late requirements analysis to prototype testing.

The Modeling Stage is mainly role-driven: role modeling allows the MAS developer to specify what the system can do, without going into the details about how the system will do it. Roles are played by agent classes.

Once the role model is well understood, the developer needs to define how communication among entities playing different roles takes place; which roles should be assigned to which agent class; and how many instances of each agent class are required for the given application. The

language that DCaseLP provides to the user in order to cope with these aspects is UML. The tool that allows the integration of role models defined using UML into DCaseLP consists of a set of XSL configuration files that define the rules for translating XMI representations of UML diagrams into executable code.

The implementation of the MAS prototype must be coherent with the specification given in the previous step. Ensuring this coherence is a demanding task for the developer of the prototype, but DCaseLP may reduce this burden by providing a semi-automatic translator from UML into Jess, that is one of the implementation languages offered by DCaseLP. During the implementation stage, the integration of external software comes into play. To this aim, DCaseLP follows the Wrapper Agent model, defined by the Foundation for Intelligent Physical Agents, FIPA, with agents that act as "wrappers" for the external pieces of code. The external packages which can be accessed by the prototype are all the ones that Java, tuProlog and Jess provide interfaces for. DCaseLP provides a set of libraries for integrating in JADE agents whose behavior is entirely programmed in tuProlog or Jess, and not simply a way of executing tuProlog or Jess pieces of code from inside JADE agents.

The execution of the MAS prototype allows the MAS developer to test and evaluate the analysis, design, and implementation choices that were made during development. DCaseLP libraries provide no direct support to MAS testing, which exploits monitoring and debugging tools offered by JADE.

2.3.1.2 Integrating other languages

DCaseLP has been implemented in order to provide

1. a *support to the steps* described in Section 2.3.1.1, and
2. a *transparent integration* between Java, Jess, and tuProlog agents running in a JADE platform (Figure 6).

DCaseLP is structured in three main packages, each devoted to manage a different language:

1. the Java $UMLInJADE$ package contains the Java classes and the xls style sheets for translating UML diagrams created with any UML modelling tool and exported into xmi, into (ad-hoc) intermediate XML models and, from these, to create the files containing the code for running the Jess agents into JADE;
2. the Java $jessInJADE$ package contains the classes that implement Jess agents, to be run in the JADE environment, and whose behavior is fully specified by means of the Jess language;

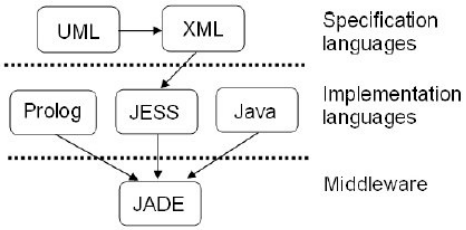


Figure 6: DCaseLP packages.

3. the Java $tuPInJADE$ package contains the classes that implement tuProlog agents, to be run in the JADE environment, and whose behavior is fully specified by means of the tuProlog language.

The three packages, together with their manuals and tutorials, are available from

`http://www.disi.unige.it/person/MascardiV/Software/DCaseLP.html`.

Examples of use of DCaseLP are also provided from the above URL.

2.3.2 JADE

Considering the platforms used to implement Multiagent systems, we describe in this section the one we used, JADE, that is also one of the most used and supported platform in this research field.

Since in the next chapters we will describe two systems implemented with JADE, the reader needs to have some basic knowledge of it, in particular of its architectural organization, of the $Agent$ class it provides, and so on. To let everybody understand the explanation of the system and the JADE simplified code we report in this section a brief description of the JADE platform, summarizing its manuals [120].

JADE (Java Agent DEvelopment Framework) is a software framework fully implemented in Java, is a free software and is distributed by Telecom Italia, the copyright holder, in open source software under the terms of the LGPL. It simplifies the implementation of multiagent systems through a middle-ware that claims to comply with the FIPA [55] specifications and through a set of tools that supports the debugging and deployment phases. The agent platform can be distributed across machines (which do not even need to share the same OS) and the configuration can be controlled via a remote GUI. The only system requirement is the Java Run Time version 5 or later.

The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent. Agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model has been implemented and its components have been clearly separated and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol.

Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behavior of each state of the protocol. SL (Semantic Language) and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.

JADE is being used by a number of companies and academic groups, such as BT, Telefonica,

CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS, and many others.

JADE is distributed in Open Source under the LGPL License. Further details and documentation can be found at http://jade.tilab.com/.

2.3.2.1 Architecture

JADE architecture includes:

- A runtime environment where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
- A library of classes that programmers have to/can use (directly or by specializing them) to develop their agents.
- A suite of graphical tools that allows administrating and monitoring the activity of running agents.
- A FIPA-compliant Agent Platform, which includes the AMS (Agent Management System) and the DF (Directory Facilitator). These components are automatically activated at the agent platform start-up.

Each running instance of the JADE runtime environment is called a Container as it can contain several agents. The set of active containers is called a Platform. A single special Main container must always be active in a platform.

The Agent Management System (AMS) is the agent who exerts supervisory control over access to and use of the Agent Platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.

The Directory Facilitator (DF) is the agent who provides the default yellow page service in the platform.

The DF and the AMS agents are foreseen in the Reference architecture of a FIPA Agent Platform.

When a JADE platform is launched, the AMS and DF are immediately created.

2.3.2.2 The $Agent$ Class

The $Agent$ class represents a common base class for user defined agents. Therefore, from the programmers point of view, a JADE agent is simply an instance of a user defined Java class

that extends the base $Agent$ class. This implies the inheritance of features to accomplish basic interactions with the agent platform (registration, configuration, remote management, ...) and a basic set of methods that can be called to implement the custom behavior of the agent (e.g. send/receive messages, use standard interaction protocols, register with several domains, ...).

The computational model of an agent is multitask, where tasks (or behaviors) are executed concurrently. Each functionality/service provided by an agent should be implemented as one or more behaviors. A scheduler, internal to the base $Agent$ class and hidden to the programmer, automatically manages the scheduling of behaviors.

The $setup()$ method is the point where any application-defined agent activity starts. The programmer has to implement the $setup()$ method in order to initialize the agent.

A list of arguments can be passed to an Agent and they can be retrieved by calling the method *Object[] getArguments()*.

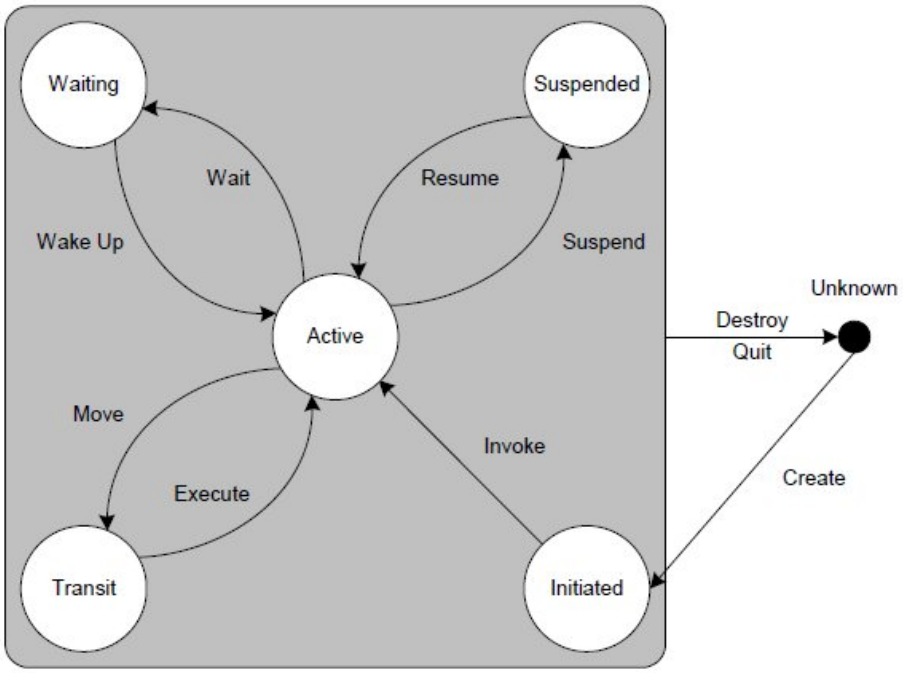


Figure 7: Agent life cycle

A JADE agent can be in one of several states, according to Agent Platform Life Cycle in FIPA specification; these are represented in Figure 7 and are detailed below:

- INITIATED : the Agent object is built, but it has not registered itself yet with the AMS, it has neither a name nor an address and it cannot communicate with other agents.

- ACTIVE : the Agent object is registered with the AMS, it has a regular name and address and it can access all the various JADE features.
- SUSPENDED : the Agent object is currently stopped. Its internal thread is suspended and no agent behavior is being executed.
- WAITING : the Agent object is blocked, waiting for something. Its internal thread is sleeping on a Java monitor and will wake up when some condition is met (typically when a message arrives).
- DELETED : the Agent is definitely dead. The internal thread has terminated its execution and the Agent is no more registered with the AMS.
- TRANSIT: a mobile agent enters this state while it is migrating to the new location. The system continues to buffer messages that will then be sent to its new location.

2.3.2.3 Inter-agent communication

The $Agent$ class also provides a set of methods for inter-agent communication. The two most well-known attempts to define a standard for specifying the structure of a complex message (with a content, a reference ontology, a performative and so on) are KQML [51] and FIPA ACL [54]. JADE complies with the FIPA specification. According to it, agents communicate via asynchronous message passing, where objects of the ACLMessage class are the exchanged payloads.

The $Agent.send()$ method allows to send an ACLMessage. The value of the receiver slot holds the list of the receiving agent IDs (that are the identification numbers inside the Platform). The method call is completely transparent to where the agent resides, i.e., be it local or remote: the platform takes care of selecting the most appropriate address and transport mechanism. The platform puts all the messages received by an agent into the agents private queue.

Several access modes have been implemented in order to get messages from this private queue:

- The message queue can be accessed in a blocking (using $blockingReceive()$ method) or non-blocking way (using $receive()$ method). The blocking version must be used very carefully because it causes the suspension of all the agent activities and in particular of all its Behaviors. The non-blocking version returns immediately $null$ when the requested message is not present in the queue;
- both methods can be augmented with a pattern-matching capability where a parameter is passed that describes the pattern of the requested ACLMessage.

2.3.2.4 Agent Communication Language (ACL) Messages

The class ACLMessage represents ACL messages that can be exchanged between agents. It contains a set of attributes as defined by the FIPA specifications:

- Performative: What action the message performs
- Sender: Initiator of the message
- Receiver: Recipient of the message
- Reply-to: Recipient of the message reply
- Content: Content of the message
- Language: Language used to express content
- Encoding: Encoding used for content
- Ontology: Ontology context for content
- Protocol: Protocol that the message belongs to
- Conversation-id: Conversation that the message belongs to
- Reply-with: Reply with this expression
- In-reply-to: Action to which this is a reply
- Reply-by: Time to receive reply by

An agent willing to send a message should create a new ACLMessage object, fill its attributes with appropriate values, and finally call the method Agent.send(). Likewise, an agent willing to receive a message should call $receive()$ or $blockingReceive()$ methods, both implemented by the $Agent$ class.

This class also defines a set of constants that should be used to refer to the FIPA performatives, i.e. REQUEST, INFORM, etc. When creating a new ACLMessage object, one of these constants must be passed to ACLMessage class constructor, in order to select the message performative.

According to FIPA specifications, a reply message must be formed taking into account a set of well-formed rules, such as setting the appropriate value for the attribute in-reply-to, using the same conversation-id, etc. JADE helps the programmer in this task via the method $createReply()$ of the ACLMessage class. This method returns a new ACLMessage object that is a valid reply to the current one. Then, the programmer only needs to set the application specific communicative act and message content.

Furthermore, the programmer can use the $MessageTemplate$ class, which allows to build patterns to match ACL messages against. Using the methods of this class the programmer can create one pattern for each attribute of the ACLMessage. Elementary patterns can be combined with AND, OR and NOT operators, in order to build more complex matching rules. In such a way, the queue of incoming ACL messages can be accessed via pattern-matching rather than FIFO.

2.3.2.5 The agent tasks. Implementing Agent behaviors

An agent must be able to carry out several concurrent tasks in response to different external events. In order to make agent management efficient, every JADE agent is composed of a single execution thread and all its tasks are modeled and can be implemented as Behavior objects.

The developer who wants to implement an agent-specific task should define one or more Behavior subclasses, instantiate them and add the behavior objects to the agent task list. The $Agent$ class exposes two methods, $addBehavior(Behavior)$ and $removeBehavior(Behavior)$, which must be extended by agent programmers and that allow to manage the ready tasks queue of a specific agent. Notice that behaviors and sub-behaviors can be added whenever is needed, and not only within the $Agent.setup()$ method.

A scheduler, implemented by the base $Agent$ class and hidden to the programmer, carries out a round-robin non-preemptive scheduling policy among all behaviors available in the ready queue, executing a Behavior-derived class until it will release control (this happens when $action()$ method returns). If the task relinquishing the control has not yet completed, it will be rescheduled the next round. A behavior can also block, waiting for a message to arrive. In detail, the agent scheduler executes the $action()$ method of each behavior present in the ready behaviors queue; when $action()$ returns, the method $done()$ is called to check if the behavior has completed its task. If so, the behavior object is removed from the queue.

Behaviors work just like co-operative threads, but there is no stack to be saved. Therefore, the whole computation state must be maintained in instance variables of the Behavior and its associated Agent.

Besides, since no stack context is saved, every time the $action()$ method is run from the beginning: there is no way to interrupt a behavior in the middle of its $action()$, yield the CPU to other behaviors and then start the original behavior back from where it left.

Remember that when some behaviors $action()$ is running, no other behavior can go on until the end of the method (of course this is true only with respect to behaviors of the same agent: behaviors of other agents run in different Java threads and can still proceed independently).

When dealing with complex agent behaviors (as agent interaction protocols) using explicit state variables can be cumbersome; so JADE also supports a compositional technique to build more complex behaviors out of simpler ones.

JADE offers the following abstract behaviors classes:

- class $Behavior$ provides an abstract base class for modeling agent tasks;
- class $SimpleBehavior$ models simple atomic behaviors;
- class $OneShotBehavior$ models atomic behaviors that must be executed only once and cannot be blocked;
- class $CyclicBehavior$ models atomic behaviors that must be executed forever;
- class $WakerBehavior$ implements a one-shot task that must be executed only once just after a given timeout is elapsed;
- class $TickerBehavior$ implements a cyclic task that must be executed periodically;
- class $CompositeBehavior$ models behaviors that are made up by composing a number of other behaviors (children). So the actual operations performed by executing this behavior are not defined in the behavior itself, but inside its children while the composite behavior takes only care of children scheduling according to a given policy. In particular the $CompositeBehavior$ class only provides a common interface for children scheduling, but does not define any scheduling policy. This scheduling policy must be defined by subclasses:
  - class $SequentialBehavior$: a $CompositeBehavior$ that executes its sub-behaviors sequentially and terminates when all sub-behaviors are done.
  - class $ParallelBehavior$: a $CompositeBehavior$ that executes its sub-behaviors concurrently and terminates when a particular condition on its sub-behaviors is met.
  - class $FSMBehavior$: a $CompositeBehavior$ that executes its children according to a Finite State Machine defined by the user.

As mentioned above, behavior scheduling is performed in a non-preemptive way, i.e., the $action()$ method of a behavior is never interrupted to allow other behaviors to go on. Only when the $action()$ method of the currently running behavior returns, the control is given to the next behavior. This approach has several advantages in terms of performances and scalability. However, when a behavior needs to perform some blocking operations it actually blocks the whole agent and not only itself. A possible solution to that problem is of course using normal Java threads. JADE however provides a cleaner solution by means of threaded behaviors i.e. behaviors that are executed in dedicated threads.

Any JADE Behavior (composite or simple) can be executed as a threaded behavior by means of the $jade.core.behaviors.ThreadedBehaviorFactory$ class. This class provides the $wrap()$ method that actually wraps a normal JADE Behavior into a $ThreadedBehavior$. Adding that

$ThreadedBehavior$ to the agent by means of the $addBehavior()$ method as usual results in executing the original Behavior in a dedicated thread. It should be noticed that developers only deal with the $ThreadedBehaviorFactory$ class, while the $ThreadedBehavior$ class is private and not accessible.

2.3.3 NetLogo

Multiagent systems are often used to simulate natural and social phenomena, where many agents (usually, hundred) interact using quite simple rules: the aim of these simulations is to analyze the complex emerging behavior of the system. In this research area one of the most used tool is NetLogo.

In this section we quote the NetLogo homepage [123] and we report the main features of this developing environment, that we used in one of our system, as shown in Section 5.5.

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It was authored by Uri Wilensky in 1999 and has been in continuous development ever since at the Center for Connected Learning and Computer-Based Modeling.

NetLogo is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of "agents" all operating independently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals.

It has extensive documentation and tutorials and also comes with a Models Library, which is a large collection of pre-written simulations that can be used and modified. These simulations address many content areas in the natural and social sciences, including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. Several model-based inquiry curricula using NetLogo are currently under development.

NetLogo is the next generation of the series of multi-agent modeling languages that started with StarLogo [37]. It builds off the functionality of StarLogoT and adds significant new features and a redesigned language and user interface. NetLogo runs on the Java virtual machine, so it works on all major platforms (Mac, Windows, Linux, et al). It is run as a standalone application, or from the command line. Models can be run as Java applets in a web browser.

The NetLogo world is made up of agents. Agents are beings that can follow instructions. Each agent can carry out its own activity, all simultaneously. There are four types of agents: $turtles$, $patches$, $links$, and the $observer$. $Turtles$ are agents that move around in the world. The world is two dimensional and is divided up into a grid of $patches$. Each $patch$ is a square piece of "ground" over which $turtles$ can move. $Links$ are agents that connect two $turtles$. The $observer$ does not have a location (you can imagine it as looking out over the world of $turtles$ and $patches$). When NetLogo starts up, there are no $turtles$ yet. The $observer$ can make new

turtles. *Patches* can make new *turtles* too. *Patches* can't move, but otherwise they're just as "alive" as *turtles* and the *observer* are.

Patches have coordinates. The *patch* at coordinates (0, 0) is called the origin and the coordinates of the other *patches* are the horizontal and vertical distances from this one. We call the *patch*'s coordinates *pxcor* and *pycor*. Just like in the standard mathematical coordinate plane, *pxcor* increases as you move to the right and *pycor* increases as you move up.

Turtles have coordinates too: *xcor* and *ycor*. A *patch*'s coordinates are always integers, but a *turtle*'s coordinates can have decimals. This means that a *turtle* can be positioned at any point within its *patch*; it doesn't have to be in the center of the *patch*.

Links do not have coordinates, instead they have two endpoints (each a *turtle*). *Links* appear between the two endpoints, along the shortest path possible.

Agents can be grouped in set of agents, or *agentset*. An *agentset* can contain either *turtles*, *patches* or *links*, but not more than one type at once.

An *agentset* is not in any particular order. In fact, it's always in a random order. And every time you use it, the *agentset* is in a different random order. This helps you keep your model from treating any particular *turtles*, *patches* or *links* differently from any others (unless you want them to be). Since the order is random every time, no one agent always gets to go first.

The "view" in NetLogo lets you see the agents in your model on your computer's screen. As your agents move and change, you see them moving and changing in the view. Of course, you can't really see your agents directly. The view is a picture that NetLogo automatically paints, showing you how your agents look at a particular instant. Once that instant passes and your agents move and change some more, that picture needs to be repainted to reflect the new state of the world. Repainting the picture is called "updating" the view.

NetLogo offers two updates modes, "continuous" updates and "tick-based" updates. You can switch between NetLogo's two view update modes using the NetLogo Interface.

Continuous updates are the default when you start up NetLogo or start a new model. Nearly every model in the Models Library, however, uses tick-based updates. Continuous updates are simplest, but tick-based updates give you more control over when and how often updates happen.

It's important exactly when an update happens, because when updates happen determines what you see on the screen. If an update comes at an unexpected time, you may see something unexpected (perhaps something confusing or misleading).

It's also important how often updates happen, because updates take time. The more time NetLogo spends updating the view, the slower your model will run. With fewer updates, your model runs faster. In many NetLogo models, time passes in discrete steps, called "ticks". Typically, you want the view to update once per tick, between ticks. That's the default behavior with tick-based updates.

2.3.4 Multiagent systems standards

As described in the previous sections, the research on Multiagent systems was born from many different research fields and nowadays this is still an open research area. As research improved and MASs started to be used even outside the boundaries of academia, the need for some commonly accepted standards emerged.

Many standards were proposed and exist but surely those most accepted and used are the ones coming from FIPA (Foundation for Intelligent Physical Agents) [55], that is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. FIPA was officially accepted by the IEEE as its eleventh standards committee on 8 June 2005. It was originally formed as a Swiss based organization in 1996 to produce software standards specifications for heterogeneous and interacting agents and agent based systems.

Since its foundations, FIPA has played a crucial role in the development of agents standards and has promoted a number of initiatives and events that contributed to the development and uptake of agent technology. Furthermore, many of the ideas originated and developed in FIPA are now coming into sharp focus in new generations of Web/Internet technology and related specifications.

2.4 Semantic issues in MAS

In a MAS different kinds of agents can enter and participate to the protocol that leads the MAS: at design time, it is crucial to understand if the MAS will be concretely "open", that is, it will admit agents developed by a third entities, or not. If unknown agents can enter the MAS they will surely need to understand the meaning of the exchanged messages, of the eventual rules and so on. Otherwise, if the MAS is "close", all the agents have been designed and developed by the same entity and in this case they will have no problem to understand each others, because they share a common "semantic", an interpretation of messages.

Consequently in "open" MAS we need to formalize the "semantics" of the exchanged messages in a format that is understandable by any agent. A possible solution of this problem is to use an "ontology".

2.4.1 What is an ontology?

The term "ontology" comes from the field of philosophy that is concerned with the study of being or existence. In philosophy, one can talk about an ontology as a theory of the nature of existence

(e.g., Aristotles ontology offers primitive categories, such as substance and quality, which were presumed to account for All That Is).

In computer science the definition given by Tom Gruber in 2008 [63] is one of the most commonly accepted:

> *In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse.*
>
> *The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members).*
>
> *The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.*
>
> *In the context of database systems, ontology can be viewed as a level of abstraction of data models, analogous to hierarchical and relational models, but intended for modeling knowledge about individuals, their attributes, and their relationships to other individuals.*
>
> *Ontologies are typically specified in languages that allow abstraction away from data structures and implementation strategies; in practice, the languages of ontologies are closer in expressive power to first-order logic than languages used to model databases.*
>
> *For this reason, ontologies are said to be at the "semantic" level, whereas database schema are models of data at the "logical" or "physical" level.*
>
> *Due to their independence from lower level data models, ontologies are used for integrating heterogeneous databases, enabling interoperability among disparate systems, and specifying interfaces to independent, knowledge-based services.*
>
> *In the technology stack of the Semantic Web standards, ontologies are called out as an explicit layer. There are now standard languages and a variety of commercial and open source tools for creating and working with ontologies.*

As Gruber underlined, there are some differences between Ontologies and Database:

- Purpose
  - Ontologies: Sharing, reuse, defining semantics.
  - Databases: storing large amounts of structured data.
- Expressive power of modeling languages
  - Ontologies: close first order logic; description logics.

  - Databases: not as powerful as First Order Logic.

- Standards for making queries
  - Ontologies: no standards, few proposals, for example SPARQL [128].
  - Databases: consolidated standards, for example SQL.

- Management systems
  - Ontologies: Protégé [9], Jena [4] (not commercial).
  - Databases: Many solid, commercial tools and environments (Oracle, etc).

2.4.2 How do we use them?

As we already shown, multiagent systems have become a new design-style to model complex scenarios where many entities have to communicate, interact, share files and information [135]. This kind of interaction may be hold among agents who know all about the others in the world (that is, in general, the framework) but also between entities that know nothing about each other except where to find them.

FIPA has done a great effort to create a standard to let the agents communicate in a controlled way, so every agent, regardless of its core language, structure, location and so on, should be able to find others in the framework and exchange understandable messages. But nothing was established by FIPA about the real content of the message: how can two (or more) agents know what language speaks the other one, and which terms and concepts are understood by it? They need a shared ontology. The support to create, share and access ontologies was proposed by FIPA in the "FIPA Ontology Service Specification", but nowadays it has some limitations.

2.4.3 The FIPA Ontology Agent

In 2001, FIPA presented a standard to organize and manage ontologies in a MAS [52]. The standard assumes to insert in the framework, at the same level of the Directory Facilitator Agent, a new agent called "Ontology Agent" (OA) that provides and offers to other agents services to deal with ontologies. This kind of design assumes that we handle an explicit ontology, namely an ontology that is not inserted into the code of the agents, but is stored outside of them (perhaps in some server), can be accessed by other agents, and can change during the time. Agents have to be informed about the ontology and must be able to use, and maybe change, it at run time.

In this scenario, having one or more agents that provide the services to uniformly access the ontologies in the system turns out to be extremely useful.

These "Ontology Agents" may be able to expose the follow services:

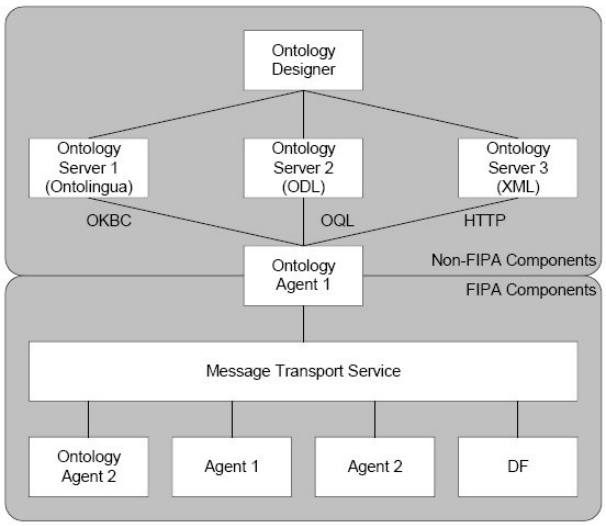


Figure 8: Ontology Service Reference Model

1. discover public ontologies in order to access them;
2. maintain (for example, to register with the Directory Facilitator – DF –, upload, download, and modify) a set of public ontologies;
3. translate expressions between different ontologies and/or different content languages;
4. answer queries about relationships between terms or between ontologies;
5. facilitate the identification of a shared ontology for communication between two agents.

It's not mandatory for an Ontology Agent (OA) to be able to realize all the services, but it must be able to answer at least that it can't provide that service. The reference model for the services provided by the Ontology Agent is shown in Figure 8.

Responsibilities, actions and predicates supported by the Ontology Agent compose the "FIPA-Ontol-Service-Ontology" ontology, that identifies the set of actions that can be requested to be performed by an OA.

FIPA provides specifications at the level of the Agent Communication Language (ACL). It does not specify how communication between the OAs and the ontology servers (for example, OKBC [5], OQL [6] or any other proprietary protocol) should take place. Therefore, a FIPA compliant OA will have to be developed on a custom basis to support interfaces with non-FIPA compliant ontology servers.

The FIPA standard proposes a Meta-ontology to describe the services of the OA; in particular, the agents that want to speak with the OA have to refer to this meta-ontology. When an agent wants to modify a particular ontology, it has to request this action to the OA using only the terms and actions from this ontology.

The definition of the meta-ontology represents the main limitation of the FIPA OA standard. In fact, FIPA assumes that each ontology integrated withing the MAS adheres to the OKBC model. The adopted model supports an object-oriented representation of knowledge and provides a set of representational constructs commonly found in object-oriented knowledge representation systems (KBS). This standard is widely used in the KBS, but differs from those used in the semantic web. In that area, OWL [7] is the most used model, and it is quite different from OKBC.

That is, if the ontology shared in the framework has OKBC type there is no problem to use the FIPA meta-ontology, but if the ontology is defined using OWL, that is almost for all ontology from the web, than there are problems to convert the operations associated with an OWL-ontology into OKBC operations. Since the two visions of data of OWL and OKBC are different, the meta-ontology is not enough to model the operations on OWL ontology.

So the FIPA standard fits only a part of the real ontologies that may be used, namely only those written in OKBC or an equivalent model. It lacks support for those OWL based or equivalent.

Another limitation of the standard is that in the ACL message an agent can specify only one ontology to refer to, but actually it is possible that in a conversation agents want to use more ontologies, and this is not allowed.

Chapter 3

MAD system: the first industrial case study

We started the research on Negotiation in Multiagent systems from an Ansaldo STS case study: Ansaldo STS (a Finmeccanica company) is the Italian leader (and also an international leader) in design and construction of signaling and automation systems for conventional and high speed railway lines. The company asked us to design a system to supervise a set of software processes running on different personal computers, all connected by a wired network.

This chapter describes this first MAS developed with Ansaldo STS, the MAD system. This case study was the first complex example of a resource allocation problem solved exploiting the agents technology analyzed during this thesis. The aim of this chapter is to completely describe this system, from the design to the implementation.

As we know from many research studies, distributed diagnosis and monitoring represent one of the oldest application fields of declarative software agents. For example, ARCHON (ARchitecture for Cooperative Heterogeneous ON-line systems [72]) was Europe's largest ever project in the area of Distributed Artificial Intelligence and exploited rule-based agents. In [106], Schroeder et al. describe a diagnostic agent based on extended logic programming. Many other MASs for diagnosis and monitoring based on declarative approaches have been developed in the past [77, 107, 131, 18]. One of the most important reasons for exploiting agents in the monitoring and diagnosis application domain is that an agent-based distributed infrastructure can be added to any existing system with minimal or no impact over it. Agents look at the processes they must monitor, be they computer processes, business processes, chemical processes, by "looking over their shoulders" without interfering with their activities. The "no-interference" feature has an enormous importance, since changing the processes in order to monitor them would be often unfeasible. Also the increase of situational awareness, essential for coping with the situational complexity of most large real applications, motivates an agent-oriented approach. Situational awareness is a mandatory feature for the successful monitoring and decision making in many scenarios. When combined with reactivity, situatedness may lead to the early detection of, and

reaction to, anomalies. The simplest and most natural form of reasoning for producing diagnoses starting from observations is based on rules.

So we chose to model the requested system as a Multiagent system (that has been called "Monitoring And Diagnostic Multiagent System" (MAD MAS)), and we decided to look at the processes and machines as interconnect resources (every process runs on a specific machine, and machines are connected by the network): the MAS must control and supervise their execution and performances. Among all the possible organizations of a MAS discussed in 2.1.2, we chose a simple hierarchical organization that reflects the physical organization of processes and machines. Hence, the coordination protocol for the agents in the MAS is a hierarchical one.

This was a first approach to agent coordination that gave us the opportunity to start learning about MARA problems, by facing one of the simplest ones, namely a problem where coordination can be structured as a hierarchy.

It also showed an industrial interest in this type of academical research and established the basis for a profitable collaboration between Ansaldo STS and our research group, collaboration that is still ongoing at the time of writing and that allowed us to design and develop a much more complex and challenging negotiation protocol named "FYPA". The application that integrates the FYPA protocol, discussed in Chapters 4, 5, and 6, was in fact another case study provided by Ansaldo STS.

3.1 The Domain and the Problem

The Command and Control System for Railway Circulation ("Sistema di Comando e Controllo della Circolazione Ferroviaria", SCC) is a framework project for the technological development of the Italian Railways ("Ferrovie dello Stato", FS), with the following targets:

- introducing and extending automation to the command and control of railway circulation over the principal lines and nodes of the FS network
- moving towards innovative approaches for the infrastructure and process management, thanks to advanced monitoring and diagnosis systems
- improving the quality of the service offered to the FS customers, thanks to a better regularity of the circulation and to the availability of more efficient services, like delivery of information to the customers, remote surveillance, and security

The SCC project is based on the installation of centralized Traffic Command and Control Systems, able to remotely control the plants located in the railway stations and to manage the movement of trains from the Central Plants (namely, the offices where instances of the SCC system

are installed). In this way, Central Plants become command and control centers for railway lines that represent the main axes of circulation, and for railway nodes with high traffic volumes, corresponding to the main metropolitan and intermodal nodes of the FS network.

An element that strongly characterizes the SCC is the strict coupling of functionalities for circulation control and functionalities for diagnosis and support to upkeep activities, with particular regard to predictive diagnostic functionalities aimed to enable on-condition upkeep. The SCC of the node of Genova, which will be employed as a case study for the implementation of the first MAS prototype, belongs to the first six plants developed by Ansaldo Segnalamento Ferroviario. They are independent one from the other, but networked by means of a WAN, and cover 3000 Km of the FS network. The area controlled by the SCC of Genova covers 255 km, with 28 fully equipped stations plus 20 stops.

The SCC can be decomposed, both from a functional and from an architectural point of view, into five subsystems. The MAS that we are implementing will monitor and diagnose critical processes belonging to the Circulation subsystem whose aims are to implement remote control of traffic and to make circulation as regular as possible. The two processes we have taken under consideration are Path Selection and Planner.

The Planner process is the back-end elaboration process for the activities concerned with Railway Regulation. There is only one instance of the Planner process in the SCC, running on the server. It continuously receives information on the position of trains from sensors located in the stations along the railway lines, checks the timetable, and formulates a plan for ensuring that the train schedule is respected.

A plan might look like the following: "Since the InterCity (IC) train 5678 is late, and IC trains have highest priority, and there is the chance to reduce the delay of 5678 if it overtakes the regional train 1234, then 1234 must stop on track 1 of Arquata station, and wait for 5678 to overtake it". Plans formulated by the Planner may be either confirmed by the operators working at the workstations or modified by them. The process that allows operators to modify the Planner's decision is Path Selection.

The Path Selection process is the front-end user interface for the activities concerned with Railway Regulation. There is one Path Selection process running on each workstation in the SCC. Each operator interacts with one instance of this process. There are various operators responsible for controlling portions of the railway line, and only one senior operator with a global view of the entire area controlled by the SCC. The senior operator coordinates the activities of the other operators and takes the final decisions. The Path Selection process visualizes decisions made by the Planner process and allows the operator to either confirm or modify them. For example, the Planner might decide that a freight train should stop on track 3 of Ronco Scrivia station in order to allow a regional train to overtake it, but the operator in charge for that railway segment might think that track 4 is a better choice for making the freight train stop. In this case, after receiving an "ok" from the senior operator, the operator may input its choice thanks to the in-

terface offered by the Path Selection process, and this choice overcomes the Planner's decisions. The Planner will re-plan its decisions according to the new input given by the human operator. The SCC Assistance Center, that provides assistance to the SCC operators in case of problems, involves a large number of domain experts, and it is always contacted after the evidence of a malfunctioning. The cause of the malfunctioning, however, might have generated minutes, and sometimes hours, before that the SCC operator(s) experienced the malfunctioning. By integrating a monitoring and diagnosing MAS to the circulation subsystem, we aim to equip any operator of the Central Plant with the means for early detecting anomalies that, if reported in a short time, and before their effects have propagated to the entire system, may allow the prevention of more serious problems including circulation delays.

When a malfunctioning is detected, the MAS, besides alerting the SCC operator, will always alert the SCC Assistance Center in an automatic way. This will not only speed up the implementation of repair actions, but also allow the SCC Assistance Center to be up to date with recorded traces of what happened in the Central Plant.

3.2 Design of the protocol MAD

In order to provide the functionalities required by the operating scenario, we designed a MAS (called MAD) where different agents exist and interact.

Both the architecture of the MAS, and the agent interactions taking place there, are quite simple. On the contrary, the rules that guide the behavior of agents are sophisticated, leading to implement agents able to monitor running processes and to quickly diagnose malfunctions.

For these reasons, among the languages offered by DCaseLP (as previously described 2.3.1.1), we have chosen to use tuProlog for implementing the monitoring agents, and Java for implementing the agents at the lowest architectural level, namely those that implement the interfaces to the processes.

3.2.1 MAS Architecture

The architecture of the MAS is depicted in Figure 1. There are four kinds of agent, organized in a hierarchy: Log Reader Agents, Process Monitoring Agents, Computer Monitoring Agents, and Plant Monitoring Agents.

Agents running on remote computers are connected via a reliable network whose failures are quickly detected and solved by an ad hoc process (already existing, outside the MAS). If the network becomes unavailable for a short time, the groups of agents running on the same computer can go on with their local work. Messages directed to remote agents are saved in a local buffer,

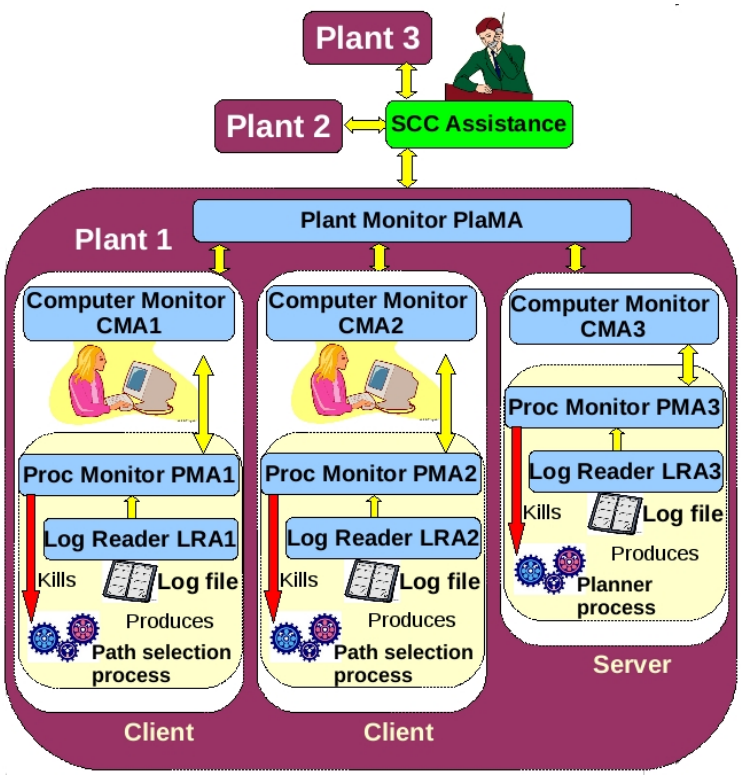


Figure 1: Architecture of the MAS

and are sent as soon as the network comes up again. The human operator is never alerted by the MAS about a network failure, and local diagnoses continue to be produced.

3.2.1.1 Log Reader Agent

In our MAS, there is one Log Reader Agent (LRA) for each process that needs to be monitored. Thus, there may be many LRAs running on the same computer.

Once every m minutes, where m can be set by the MAS configurator, the LRA reads the log file produced by the process P it monitors, extracts information from it, produces a symbolic representation of the extracted information in a format amenable of logic-based reasoning, and sends the symbolic representation to the Process Monitoring Agent in charge of monitoring P. Relevant information to be sent to the Process Monitoring Agent include loss of connection to the net and life of the process.

The parameters, and corresponding admissible values, that an LRA associated with the "Path Selection" process extracts from its log file, are:

- connection_to_server (active, lost)
- answer_to_life (ready, slow, absent)
- cpu_usage (normal, high)
- memory_usage (normal, high)
- disk_usage (normal, high)
- errors (absent, present)

The answer_to_life parameter corresponds to the time required by a process to answer a message. A couple of configurable thresholds determines the value of this parameter:

- time $<$ threshold1 implies answer_to_life ready
- threshold1 $<=$ time $<$ threshold2 implies answer_to_life slow
- time $>=$ threshold2 implies answer_to_life absent

In a similar way, the parameters and corresponding admissible values that an LRA associated with the "Planner" process extracts from its log file include:

- connection_to_client (active, lost)
- computing_time (normal, high)
- managed_conflicts (normal, high)
- managed_trains (normal, high)
- answer_to_life (ready, slow, absent)
- cpu_usage (normal, high)
- memory_usage (normal, high)
- disk_usage (normal, high)
- errors (absent, present)

LRAs have a time-driven behavior, do not perform any reasoning over the information extracted from the log file, and are neither proactive, nor autonomous. They may be considered very simple agents, mainly characterized by their social ability, employed to decouple the syntactic processing of the log files from their semantic processing, entirely demanded to the Process Monitoring Agents. In this way, if the format of the log produced by process P changes, only the LRA that reads that log needs to be modified, with no impact on the other MAS components.

3.2.1.2 Process Monitoring Agent

Process Monitoring Agents (PMAs) are in a one-to-one correspondence with LRAs: the PMA associated with process P receives the information sent by the LRA associated with P, looks for anomalies in the functioning of P, provides diagnoses for explaining their cause, reports them to the Computer Monitoring Agent (CMA), and in case kills and restarts P if necessary.

It implements a sort of social, context aware, reactive and proactive expert system characterized by rules like:

- if the answer to life of process P is slow, and it was slow also in the previous check, then there might be a problem either with the network, or with P. The PMA has to inform the CMA, and to wait for a diagnosis from the CMA. If the CMA answers that there is no problem with the network, then the problem concerns P. The action to take is to kill and restart P;
- if the answer to life of process P is absent, then the PMA has to inform the CMA, and to kill and restart P;
- if the life of process P is right, then the PMA has nothing to do.

3.2.1.3 Computer Monitoring Agent

The CMA receives all the messages arriving from the PMAs that run on that computer, and is able to monitor parameters like network availability, CPU usage, memory usage, hard disk usage. The messages received from PMAs together with the values of the monitored parameters allow the CMA to make hypotheses on the functioning of the computer where it is running, and of the entire plant. For achieving its goal, the CMA includes rules like:

- if only one PMA reported problems local to the process it monitors, then there might be temporary problems local to the process. No action needs to be taken;
- if more than one PMA reported problems local to the process it monitors, and the CPU usage is high, then there might be problems local to this computer. The action to take is

to send a message to the Plant Monitoring Agent and to make a pop-up window appear on the computer monitor, in order to alert the operator working with this computer;

- if more than one PMA reported problems due either to the network, or to the server accessed by the process, and the network is up, then there might be problems to the server accessed by the process. The action to take is to send a message to the Plant Monitoring Agent to alert it and to make a pop-up window appear on the computer monitor.

If necessary, the CMA may ask for more information to the PMA that reported the anomaly. For example, it may ask to send a detailed description of which hypotheses led to notify the anomaly, and which rules were used.

3.2.1.4 Plant Monitoring Agent

There is one Plant Monitoring Agent (PlaMA) for each plant. The PlaMA receives messages from all the CMAs in the plant and makes diagnoses and decisions according to the information it gets from them. It implements rules like:

- if more than one CMA reported a problem related to the same server S, then the server S might have a problem. The action to take is to notify the SCC Assistance Center in an automatic way;
- if more than one CMA reported a problem related to a server, and the servers referred to by the CMAs are the different, then there might be a problem of network, but more information is needed. The action to take is to ask to the CMAs that reported the anomalies more information about them;
- if more than one CMA reported a problem of network then there might be a problem of network. The action to take is to notify the SCC Assistance Center in an automatic way;

The SCC Assistance Center receives notifications from all the plants spread around Italy. It may cross-relate information about anomalies and occurred failures, and take repair actions in a centralized, efficient way.

3.3 Implementation of the MAS

All the agents of the MAS, apart from LRA that is a pure JADE [120] agent, have been implemented in tuProlog [43] integrated into JADE by means of an extended version of DCaseLP libraries [89]. The extension consists in making a *blocking_selective_receive* predicate available

to the agents, which takes three arguments: *Performative*, *Content*, *Sender*. It was motivated by the need to allow our agents to retrieve only messages respecting a given pattern (in particular, messages arriving from a given *Sender*) from their message queue. JADE offers the *MessageTemplate* class that provides static methods to create filters for each attribute of the *ACLMessage*. The JADE *blockingReceive* method can accept a message template as argument, and retrieve only those messages that match the template. The DCaseLP *blocking_selective_receive* predicate creates a template that filters on the name of the *Sender*, and then calls the JADE *blockingReceive(mt)* to return the value for *Performative* and *Content*.

LRAs have been designed and developed as agents for clearly separating what has been developed as part of this project ("agents") from what already existed ("non agents"). We also wanted to emphasis their autonomy (although very limited) and to separate the functionality of parsing the log-file from the one of reasoning over facts. However, LRAs are very trivial agents and we could have designed and implemented them as "Artifacts" in the A&A metamodel [104] or as "Touchpoints" in the Autonomic computing terminology [92] as well.

The CMA, PMA and PlaMA have a cyclic "observe-think-act" behavior [75] (and a "cyclic behavior" in JADE) where they

- look if a new message matching a given template has been received;
- retrieve the message from their message queue and store it in their history;
- manage the message according to the rules in their program, and to their knowledge base (that includes all the messages received in the past);
- answer to the agent that has sent the message, and, in case, send messages to other agents in the MAS.

The architecture of each agent, apart LRA ones, is a declarative architecture where the knowledge base is modeled as a set of Prolog facts, the behavior is determined by Prolog rules, reactivity is implemented by allowing agents to look at their message box and to react to incoming messages. Messages arrive from the LRA to the PMA every m seconds (where m is a configuration parameter of the MAS), and the PMA looks for anomalies and starts the managing process if necessary.

Agents are equipped with different rules dealing with the different parameters to be monitored, namely:

1. parameters tightly connected to the process monitored by the PMA; these parameters include "cpu_usage" and "errors" and are not influenced by the state of the network or by other processes;

2. parameters influenced either by the state of the network, or by the behavior of other processes as those running on the server (for example, “connection_to_server” and “view”).

Parameters of the first type are treated locally by the PMA. Parameters of the second type are dealt with by PMA asking the CMA, which can ask the PlaMA, for more information, since they may involve non-local problems.

An example of message sent by the LRA to the PMA is:

```
log(time(``Mon Feb 11 21:30:43 CET 2008''),
[view(normal), cpu\_usage(normal),connection\_to\_server(active),
disk\_usage(normal), answer\_to\_life(slow),
errors(absent), memory\_usage(normal)])
```

whose meaning is easy to understand.

Currently the agents do not use a common ontology, that is implicitly known as the set of the monitored parameters and their possible values.

The state of an agent consists of a set of facts representing what happened in the past. Different agents store different facts: PMAs store information about what local problems have been found and when (facts reporting a timestamp and what the problem is), CMAs keep information about the problems of all its PMAs and the notifications of a process killing (facts reporting the name of the process, a timestamp and what the problem is and facts reporting why and when a process have been killed), whereas PlaMA records facts about problems in the network (facts reporting the name of the machine and the process, a timestamp and what the problem is), but nothing about the solutions that have been taken (because they are local solutions). Messages received in previous interactions are also stored by agents in their knowledge bases, since agents may act in different ways if some problem is reported for the first time or if the problem is common to other agents that recently reported it.

This structure allows us to leave the rules that establish how to manage a problem (kill o not, according to the CMA advice) in the PMA, to store the intelligence to monitor a computer and decide when more information is needed in the CMA, and to have the PlaMA look over the whole network and answer CMAs’ requests, but without intruding in the local management.

In the sequel we show some schemes of interaction protocols among agents aimed at managing some parameters. The translation from these schemes into Prolog code has been done creating and distributing rules among the agents involved in the interaction. We did not use any standard interaction protocol: indeed, we designed the needed interaction protocols on an application-driven basis.

For example, in order to manage the "connection_to_server" parameter, the MAS acts in the following way:

1. If the value of the "connection_to_server" parameter received by the PMA from the LRA is "active", no action has to be taken; instead
2. if the value received by the PMA from the LRA is "lost", the PMA asks the CMA to know if this problem is common to other processes or not.
   (a) If the CMA has no recent information[1] about this problem in its history, it notifies the PMA that the problem is not a "net problem" (that is, it is not common to other processes); in this case, the PMA kills and restarts the process, and informs CMA of this.
   (b) If the CMA has other (one or more) recent notifications of the problem, before answering to the PMA, it asks the PlaMA to know if the problem is local to the machine where the CMA runs, or has been reported also on other computers.
      i. If the PlaMA received no notifications of this problem from other CMAs recently, it answers that the problem is not common to the MAS; in this case, the CMA answers the PMA that there are no network problems, and the PMA kills and restarts the process and informs CMA of this. Otherwise,
      ii. if the PlaMA already received notifications of the same problem in the last M minutes, it answers the CMA that there are network problems; the CMA forwards this answer to the PMA, which, in this case, does not kill the process.

There are also situations where the PMA waits for two (or more) consecutive messages from the LRA notifying the same problem, before reporting it to the CMA. This situation is shown below, for the parameter "answer_to_life":

1. If the value of the "answer_to_life" parameter received by the PMA from the LRA is "ready", no action has to be taken.
2. If the value received by the PMA from the LRA is "absent", the PMA kills and restarts the process (and informs CMA).
3. If the value received by the PMA is "slow", the PMA must wait for the successive message arriving from the LRA. If the successive message reports again a "slow" answer to life, then the PMA must ask the CMA to know if this problem is common to other processes or not.

[1] With "recent information" we mean information stored no later than M minutes ago, where M is a parameter that can be set by the person in charge of configuring the MAS.

(a) If the CMA received no recent notifications of this problem, it notifies the PMA that the problem is not a "net problem" (that is, it is not common to other processes). In this case, the PMA waits for two more messages from the LRA and, if the problem persists, kills and restarts the process and notifies it to CMA.

(b) If the CMA received recent notifications of the same problem, it asks the PlaMA if the problem is local to the machine or had also been reported on other computers.

   i. If the PlaMA answers that the problem is not common to the MAS, since no notifications of the same problem have been reported in the last M minutes, the CMA sends a "no net problem". The PMA waits for two next messages and kills and restarts the process (and notifies CMA), if the problem persists.

   ii. If the PlaMA was recently notified of the same problem, it answers the CMA that there are network problems; this answer is forwarded by the CMA to the PMA, that does not kill the process.

The hierarchical structure of the system ensures scalability: there will always be only one PlaMA in the MAS, but many CMAs may be connected to it, and many PMAs can be started on a machine and controlled by the local CMA. In case other PMAs should be developed in the future, with rules ad hoc for different processes, only the new PMAs and their associated LRAs should be developed from scratch, with no impact on the entire system.

3.4 Examples

In order to run the developed system, JADE and tuProlog (version 1.3, in order to be compliant with the DCaseLP libraries) need to be installed on the machine, as well as the extended DCaseLP libraries. The simplest configuration of the MAS includes

- one PlaMA
- one CMA
- one PMA

but usually the MAS will consist of at least two CMAs controlling different PMAs. When we tested the project we used more PMAs of the same type, which was not a problem because the rationale was to simulate the behavior of the CMA with more processes, regardless of their type. The PlaMA was one for each MAS. In the sequel we show the behavior of the MAS concerning the management of different parameters, and with different configurations and history. Some figures will not show the LRA to let the reader better understand the interactions among the other agents.

The first example shows the behavior of the MAS when the value “high” of the “cpu_usage” is reported by the LRA to the PMA, with the simplest MAS configuration consisting of just one agent of any kind.

When the PMA receives a message from the LRA:

1. If the value of the “cpu_usage” parameter is “normal”, no action needs to be taken.
2. If the value of the parameter is “high”, and it remains high in the successive message sent by the LRA, the PMA kills and restarts the process, and informs the CMA.

The simplest MAS configuration works well enough to demonstrate this behavior, because it does not depend on how many PMAs encountered the same problem. As shown in Figure 2, the first message notifying a high cpu usage from the LRA does not cause the delivery of message from the PMA. The second message with the same content, instead, causes the PMA to send a message to the CMA, with the content “process killed”.

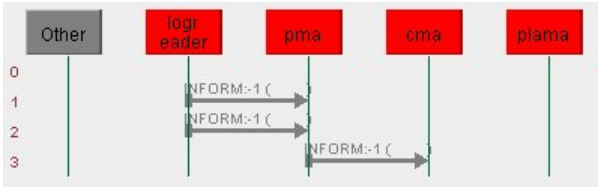


Figure 2: Execution run concerning the “cpu_usage” parameter.

Figure 3 depicts the behavior of the PMA with respect to the “answer_to_life” parameter. In this configuration we have only one CMA. The PMA will wait for two messages from the LRA with the value “slow” for the parameter, then it contacts the CMA for further information: CMA received no recent information from other PMAs, so PMA kills (after two messages from LRA with the same problem) the monitored process and informs of this the CMA, as described in the previous section.

The behavior of the system for the management of the “connection_to_server” parameter is shown in the third example. The behavior has been illustrated in Section 3.3 and is more complex than the one dealing with the “cpu_usage”. To allow a good understanding of how it works, we will use two different configurations and histories.

The first configuration, shown in Figure 4, involves one PlaMA, one CMA and two PMAs, named Pma1 and Pma2. Pma1 receives a message from its LRA with “connection_to_server(lost)”: Pma1 asks for more information to CMA, that has no recent notifications of this problem from other PMAs, and answers “no_network_problem” to Pma1. Pma1 kills and restarts the process and informs CMA of this. Later, also Pma2 receives the same message from its LRA, and, in the same way as Pma1, asks to CMA if the same problem has already been reported. CMA, which

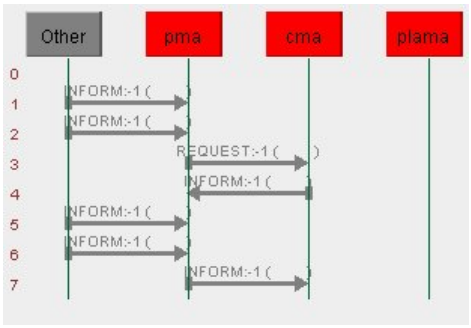


Figure 3: Execution run concerning the "answer_to_life" parameter.

had registered the problem of Pma1 in its history, needs to verify if this is a local problem or a problem involving the entire network. Thus, it asks the PlaMA if it is aware of other CMAs with the same problem. For the PlaMA, this is the first notification of the problem so it registers it into its history and answers "no_network_problem". The CMA forwards the message to Pma2 which kills and restarts the process, and informs CMA of it.

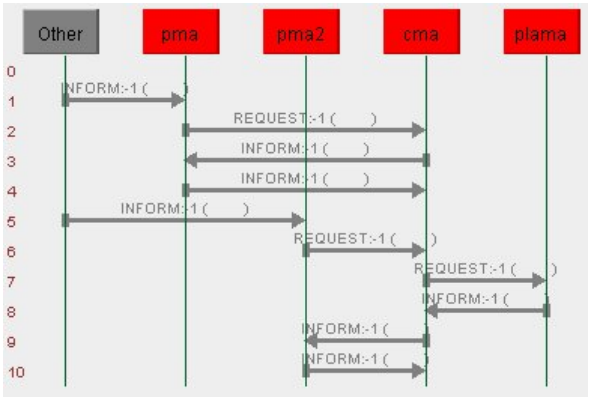


Figure 4: Execution run concerning the "connection_to_server" parameter.

If we make the configuration even more complex (Figure 5), the behavior of the MAS changes. We add another CMA named Cma2, controlling two PMAs (Pma3 and Pma4). The agents shown in Figure 4 are still alive and their history includes the events discussed before. If Pma3 receives the notification of the "connection_to_server(lost)" problem, it reacts exactly as Pma1, and Cma2 acts as Cma1. That is, Cma2 answers to Pma3, without asking the PlaMA, that there are no network problems. But if also Pma4 receives the "connection_to_server(lost)" message from its LRA, then Cma2 must ask the PlaMA if there are network problems. The PlaMA's history contains the fact that Cma1 reported the same problem a short while ago, so PlaMA sends a message with content "network_problem" to Cma2. This answer is propagated to Pma4 by Cma2, and, as a consequence, Pma4 does not kill the process because the problem cannot be managed locally.

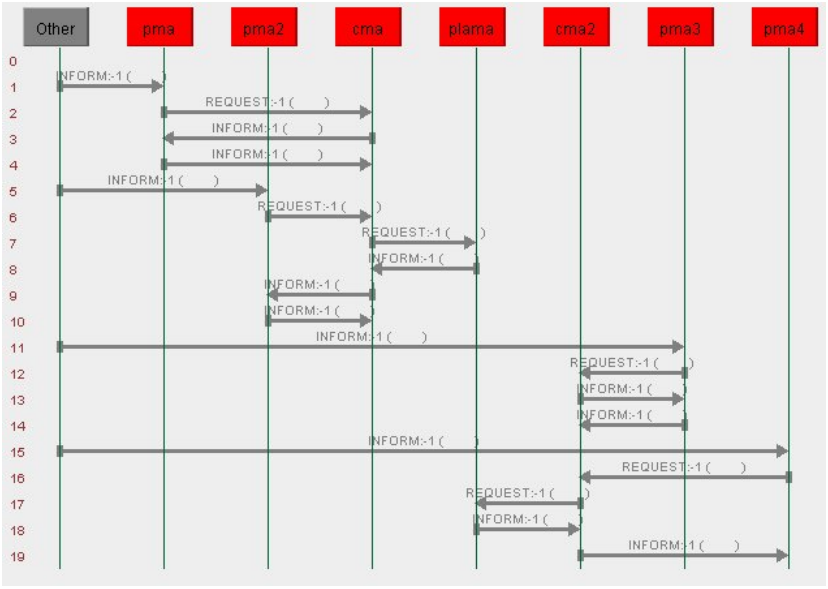


Figure 5: Execution run concerning the "connection_to_server" parameter, complex configuration.

3.5 Evaluation of the MAD system

The joint DISI-Ansaldo project confirms the applicability of MAS technologies for concrete industrial problems. Although the adoption of agents for process control is not a novelty, the exploitation of declarative approaches outside the boundaries of academia is not widespread. Instead, the Ansaldo STS partners recognized that Prolog is a suitable language for rapid prototyping of complex systems: this is a relevant result. The collaboration has led to the implementation of a first MAS prototype at the end of June 2008, to its experimentation and, in the long term perspective, to the implementation of a real MAS and its installation in both central plants and stations along the railway lines (Ansaldo STS is currently analyzing this possibility).

If Ansaldo STS will decide to move on with this project, the possibility to integrate agents that exploit statistical learning methods to classify malfunctions and agents that mine functioning reports in order to identify patterns that led to problems will be considered. Moreover, in that case more sophisticated rules will be added in the agents: for example we could identify from the log files the situation where a process has been killed and restarted a suspicious number of times.

Chapter 4

FYPA: the second industrial case study

The case study that we used as a starting point to develop the negotiation protocol described in this chapter originates from a collaboration with Ansaldo STS Genoa. This collaboration and its results are protected by a Non Disclosure Agreement but Ansaldo STS has kindly agreed on publishing some aspects of it, in particular those regarding the design and the development of the Multiagent system.

In this chapter the problem and the domain will be presented in details, and the protocol designed to solve the allocation problem will be fully analyzed. A detailed description of the developed system will be reported in the end of the chapter.

We will present the pseudo code of the main behaviors of the agents to give an overall overview of the protocol and of the system organization. Moreover, the simplified code of agents is reported in Appendixes A, B, C, D and E.

4.1 The Domain and the Problem

The problem that we faced is a typical MARA problem (Multiagent Resources Allocation Problem): we had to design a Multiagent system able to manage the real-time allocation (and re-allocation) of a set of limited resources. The resources are railway tracks (or only "railway" further on) inside a station plus a set of segments of railway where trains are allowed to stop (for example to let the passengers in and out) that we call "Stop nodes": these nodes are connected by one, or usually more, railways. A "Stop node" can be occupied by only one train at a time (no contemporary trains). Due to the physical distribution in the space of "Stop nodes", some of the railways intersect with each other so it is not possible for different trains to use them at the same time, because they may crash.

A train can enter or leave the station using only a set of "Stop nodes", that we call "Entering and Exiting nodes" respectively. Once inside the station the train can move following the railways to reach some "Stop nodes", can wait there for a period it decides and then can reach an "Exiting nodes" and leave the station. Our system manages only trains inside the station or entering it. When they reach an "Exiting nodes", they leave the station and are no more managed by our system.

Every train has a predefined path in the station that it should follow: it can change some "Stop nodes" if needed but it usually has to change as little as possible from its original plan. All the original plans of trains crossing a station are calculated from an external system that is able to assign paths without incompatibilities, that is, without crashes (more trains at the same time on the same "Stop node" or trains using incompatible railways at the same time). But some problems arise when we look at this organization at real-time: incoming trains come from other stations that we do not manage and can arrive with delays, so their original plans may not be still valid because other trains, for example, are using the "Stop nodes" they needed. Moreover, a physical failure sometimes may happen in the station, for example a railway may not answer commands or a "Stop node" may not be available because under maintenance. In this case too, the original plans that trains had to follow might not be any longer consistent with the real state of the rail station.

The main aim of our protocol is to dynamically find (in real-time) a solution to these unavailabilities, or in other words to find a new path in the station for every train, respecting all the constrains. Currently this re-allocation is made by human operators from the managing center of the station, but Ansaldo STS desired to have a decision support system capable of automatically suggesting a good solution to the user. Then the human controller can accept it or discard it.

When an allocation problem arises in the station we have to change the plan of one, or more, trains: every train has a specific typology (may be a slow passenger train, a fast one, a goods carrier and so on) and every typology has a priority. So if we have to make a train stop for more time on a "Stop node", waiting for the next node to become free (forcing in this way the train to wait), usually we prefer stopping the train with lower priority. The aim of the algorithm is also to find a solution where the delay of the involved trains is as limited as possible, and where the higher priority of a train implies a minor delay. Moreover the protocol has to change the less is possible the original plans.

As requested by Ansaldo STS, the solution to the problem must be found very quickly (within few seconds), so a sub-optimal solution is accepted.

4.2 Design of the FYPA protocol

The problem proposed by Ansaldo STS is well suited to be generalized, so in this section we will present a generic representation of the domain and of the entities participating to the protocol, to show how this case study can be seen as an example of a larger set of MARA problems. The protocol has been named "FYPA (Find Your Path, Agent!)", as suggested by Riccardo Caccia, our reference Ansaldo STS engineer.

The problem consists of:

- A set of indivisible resources that must be assigned to different entities in different time slots (each resource can be used by only one entity in each time slot).
- A set of entities with different priorities, each needing to use some of the available resources for one or more time slots; entities have preferences over the set of resources they can obtain.
- A directed graph of dependencies among resources: an entity can start using resource R only if it used exactly one resource from $\{R_1, R_2, ..., R_n\}$ in the previous time slot (we represent these dependencies as arcs $R_1 \rightarrow R$, $R_2 \rightarrow R$, ..., $R_n \rightarrow R$ in the graph).
- A set of resources named "start points" that can be assigned to entities without requiring the prior usage of other resources (no arc enters in the corresponding node).
- A set of resources named "end points" that, once assigned to one entity, allow the entity to complete its job (no arc exits from the corresponding node).
- A set of couples of conflicting arcs in the graph of dependencies: an entity releasing R_1 for accessing R_2, where the usage of R_2 depends on the previous usage of R_1, might conflict with an entity releasing R_3 for accessing R_4. The two entities might indeed need to use the same transportation means for accessing R_2 from R_1 and R_4 from R_3 respectively, and the transportation means might be non sharable as well.
- A static allocation plan that assigns resources to entities for predefined time slots, in such a way that no conflicts arise.

In an ideal world where resources never go out of order and where any entity in the system can always access the resources assigned to it by the static allocation plan, no problems arise.

In the real world where entities happen to use resources for longer than planned and where resources can break up, a dynamic re-allocation of resources over time is often required. Thus, the solution of the real world problem is a dynamic re-allocation of the resources to the entities such that:

1. the re-allocation is feasible, namely free of conflicts; in our scenario, conflicts may arise both because two or more entities would like to access the same resource in the same time slot, and because two or more entities would like to use conflicting arcs in the same time slot;

2. the re-allocation task, whose output is either a feasible re-allocation or a message stating that no feasible re-allocation exists, is completed within a predefined amount of time;

3. each entity changes the less is possible its static allocation plan: the start and end point must always remain those stated in the static allocation plan, but the nodes in between may change, as well as the time slots during which resources are used;

4. each entity admits a low delay in reaching the end point (with respect to its static allocation plan);

5. the number of entities and resources involved in the re-allocation process is kept to the minimum;

6. in case no feasible re-allocation exists, the algorithm stops.

Goals 3 and 4 are local to the entities, whereas goal 5 is a global one. Achieving these three goals together is often impossible since the best re-allocation for one entity might cause the involvement of many other entities in the process, which contrasts with the last goal. Also, the re-allocation process must output either some conflict-free plan or a "no solution exists" message in a limited and predefined amount of time (goal 2), for functioning requirements of Ansaldo-STS's application. Thus, the plan resulting from the re-allocation problem, if any, may be sub-optimal.

If we need to generalize this problem, we can model it as a directed and non-planar graph that entities must cross from one start point to one end point.

Nodes in the graph are labeled by resources (in the sequel, we will sometimes use "node" and "resource" interchangeably, identifying a node by the resource that labels it) whereas arcs represent dependencies among them. In this generalization of the protocol we adopt a discrete and linear time model (a simplification of the real domain, where the time is continuous).

The fact that an entity stops in a node R for a given time slot TS corresponds to the usage of R during TS by the entity. Arcs $R_1 \rightarrow R$, $R_2 \rightarrow R$, ..., $R_n \rightarrow R$ in the graph mean that, in order for an entity to start using R at time T, exactly one resource among $R_1, R_2, ..., R_n$ had to be used by the same entity at time $T-1$. The resource in the set $R_1, R_2, ..., R_n$ might have been used by the entity for more than one time slot, namely from $T-n$ to $T-1$, $n > 1$. We are not interested in what happened before $T-1$. The important thing is that at time $T-1$, the entity used exactly one resource in the set.

When an entity traverses an arc $R_1 \rightarrow R_2$ it releases R_1 and accesses R_2. We assume that traversing an arc always requires one time unit. This is not true in the real application that we addressed, where the traversal time may vary: this simplification does not change too much the original problem but made the problem easier to be generalized and formalized. The usage of $R_1 \rightarrow R_2$ during TS makes $R_1 \rightarrow R_2$, and all the arcs that conflict with it, occupied for TS.

Entities enter the graph one after the other: in any time instant T, there is at most one entity showing up in one "start point".

In order to keep our model as general as possible, we make the following assumptions:

1. there may be many different direct arcs connecting two nodes;
2. there may be arcs that intersect (since the graph is not planar) and that cannot be traversed concurrently by two entities.

Entities need to cross the graph from one start point to one end point. Entities cannot change the start and end points established by their static allocation plan, but they are free both to find another path connecting these points and to modify the time slots during which they use some resources, if the original path stated by the static allocation plan is no longer available. The problems that may cause an entity to change its original static allocation plan are:

1. A delay of the entities in entering the start node with respect to the original plan;
2. A node in the original path is either no longer available (because the corresponding resource is out of order), or not free in the needed time slot because another entity is using it;
3. An arc that the entity should use, to move from the current node to another node, is out of order or is not free in the needed time slot.

Given this model, the problem to solve can be stated as:

> *For each entity that enters the graph from a start point either confirm the validity of the plan stated in the static allocation plan or, if some unexpected event occurred that makes the original plan no longer applicable, find a new consisting plan for reaching the end point of the graph as stated in the original plan. The new plan should get the entity exits the graph with a low delay, modifying the original plan the less is possible, as well it should involve few entities in the re-allocation process.*

Note that all the problems may appear in any moment and more than once, affecting more than one entity during its traversal of the graph.

Every entity has the *local goal* of eventually getting out of the graph, trying to get as little delay as possible and to change as little as possible its path in the graph. Entities have a partial knowledge of the state of nodes and arcs in the graph. Entities are self-interested and try to get the control of those resources that would allow them achieving their goal. However, the system enforces a set of rules that all entities must respect in order to negotiate the control over the resources in a fair way. Entities are heterogeneous since they may have different priorities. The priority will be one of the factors that will determine how to allocate the resources, as explained in the next subsections.

The *global goal* of the system is to limit the number of entities involved in the re-allocation process.

4.2.1 Organization of the MAS: the entities participating

The problem described in Section 4.1 involves heterogeneous, autonomous, self-interested and distributed entities each with a partial knowledge of the environment where they live. Entities compete for obtaining the control of the resources needed for completing their job (i.e., exiting the graph), but they also need to cooperate in order to find a new feasible allocation that respects the rules imposed by the system (i.e., keeping the number of entities involved in the re-allocation as small as possible). These features make the dynamic resource allocation problem extremely suitable for being faced with an agent-oriented approach. Thus, we designed three different types of agents each with its own capabilities and view of the system:

- **Resource Agents (RA)**: each node in the graph is managed by one RAs in the MAS that knows the state of the node and the state of the arcs entering in it.
- **User Agents (UA)**: entities traversing the graph become UAs in the MAS.
- **Interface Agents (IA)**: agents responsible for the interactions with the environment outside the MAS are named IA.

The graph becomes the environment where agents live. There is no central control of the state of the graph, which is indeed spread all over the RAs.

4.2.1.1 Resource Agents (RA)

Each node in the graph is managed by one RA. RAs do not take decisions about which UA will obtain the control of the node but keep track of the node's state (free/occupied). RA also manage the allocation of arcs entering the node.

UAs interact with RAs to know whether the node is free or occupied in a given time slot. RAs answer the question and, in case the node is not free, tell which UA occupies the node for the given time slot, its priority, and when the node will become free again.

RAs also manage the allocation of the arcs incoming into the node they control. The RA controlling node N has the list of all its neighbors, namely those RAs controlling nodes N_{from} such that an arc $N_{from} \rightarrow N$ exists. For each arc $N_{from} \rightarrow N$, the RA also possesses the list of arcs $A_1, ..., A_k$ that conflict with $N_{from} \rightarrow N$. Note that two arcs may conflict even if they have no common nodes; conflicts between arcs may be represented as intersecting arcs within the graph which is, in fact, non-planar.

When the RA receives a reservation request for node N and chooses a free arc $N_{from} \rightarrow N$ to reach it, it updates its reservation table by marking the arc as occupied, answers the request, and informs all the RAs that may be interested by this reservation, namely those controlling arcs $A_1, ..., A_k$, about the new state of the arc. These RAs need to know that arcs conflicting with $N_{from} \rightarrow N$ cannot be used for the specified time slot.

In this way, the neighbors of RA have up-to-date information about the state of the arcs that might cause conflicts with their own arcs, and will be able to provide conflict-free answers to successive reservation requests.

4.2.1.2 User Agents (UA)

Each UA has an original plan (its static allocation plan) consisting of the list of nodes to go through, together with arrival and departure time for each of them. As soon as an UA enters the graph it tries to get a reservation for the nodes in its original path: if it is late, it will try to reserve its original path shifted in time of the delay it has. UAs do not try to reserve a specific arc to reach a node: they ask RAs to find a suitable arc for them.

UAs do not know the topology of the graph; they may interact with the Paths Manager Agent, introduced later on in this section, to obtain information on the paths that connect the start point where they enter the graph with the end point they must reach to exit the graph.

UAs communicate only with RAs, to reserve resources.

Every UA has a priority that it uses to reserve a resource or even to “steal” a resource to another UA.

Since each UA has the unique goal of getting out of the graph, it continues to look for a path in it until it obtains the reservation for all the nodes in the path. If it looses the reservation of one of these nodes, for example because a UA with higher priority stole the node, it will start the negotiation again until it will succeed in reserving all the nodes and getting out of the graph.

4.2.1.3 Interface Agents (IA)

IAs act as an interface between the MAS and the external environment. Moreover, they manage the creation of all the other agents (for this reason in the further chapters they will be also called Manager Agents). There are three types of IAs:

- The **Paths Manager Agent (PA)** offers the UAs the service of finding a new path in the graph. It knows the structure of the graph and it is able to find all the paths between two nodes, sending back a list of these paths in priority order (the priority function is out of our scope and can be, for example, the length of the path).

  In the "Ansaldo FYPA" system this agent provides an interface between agents in the MAS and the *Path Finder Service* offered by the Ansaldo software module external to the MAS: moreover, in Ansaldo's application the order of the returned list depends on the number of nodes in the path and on geometrical constraints.

  This agent will not return the arcs to reach the nodes but only the sequence of nodes of the paths. Some UAs could also want to pass across one particular "Stop node": in this case the PA will get this third parameter and will answer again with the list of possible new paths, all crossing the requested "Stop node" too. Moreover, a UA can also specify a node that it needs to avoid, so the PA will delete the paths containing this node from the answer. Lastly, the UA will also specify its original (current) path, so the PM will not return this one in the list.

- The **Resource Agents Manager** reads the structure of the graph from a configuration file that includes real data and creates the RAs corresponding to the nodes, equipped with all the information they need (entering arcs and incompatibilities), and will also manage the unavailabilities of resources, informing the interested Resource Agent.

- The **User Agents Manager** creates the UAs that enter the graph according to a configuration file and taking the real data on the agents' delay into account, if any. This agent will also answer to requests from the outside of the MAS about the reservation of a specific UA.

4.2.2 The Negotiation Algorithm

4.2.2.1 Resource reservations

The algorithm describes the interactions among agents in the graph, specifying how and when they can communicate and which rules they have to respect: in the next sections we describe the algorithm and we report the pseudo code of the main parts of it.

UAs will try to get a reservation of the resources they need. RAs will store these reservations and will help UAs to find an agreement offering them the information needed about the other UAs and about the state of the graph. Depending on the situation, a UA can make two different types of reservation for a resource:

- Disputable: This reservation can be accepted by an RA only if it is free and can be canceled ("stolen") by other UAs
- Non Disputable: This reservation must be accepted by the RA even if the resource is already reserved in a Disputable way (it must cancel the previous reservation) and can not be canceled ("stolen") by other UAs

Note that it is impossible to "stole" a Non Disputable reservation with another Non Disputable reservation, while a Disputable reservation can be stolen by any type of reservation. In the next sections we will describe this procedure more in detail.

When a UA wants to reserve a resource R that labels node N for a given period (TS), it sends a request to the RA in charge of R and indicates the arrival and departure times in R, its priority, and the node N_{from} in the graph from which it wants to reach N. The RA accesses its reservation table and may answer:

- "ok" if R is free in TS (or if the request is Non Disputable and there is no other Non Disputable reservation for R in TS);
- "occupied by UA X" if R is occupied by UA X in TS; in this case, it also sends information about X's priority and about the time when R will become free again;
- "not available" if it is not able to find a free arc from N_{from} to N or if the resource itself is not available (for example because it went out of order).

If R is free in TS, then the UA gets an "ok" message from the RA and has to confirm its reservation within a specified amount of time (every request of availability has a time out).

So a RA will manage requests with different states:

- Waiting: These are the requests that have been received by the RA but that have not yet been answered by the RA.
- Answered: These are the requests that have been received by the RA, that could be satisfied ("ok"), that have been answered by the RA, but that have not yet been confirmed by the UA.

- Confirmed: These reservations have been confirmed by the UA.
- Refused: These are the requests that have been received by the RA but that could not be satisfied ("occupied" answered by the RA)

A UA will manage the same requests plus the "not accepted" ones, that represent requests that a RA is not able to satisfy (those "occupied" or "not available").

4.2.2.2 Life cycle and rules of a User Agent

When a UA is created by the UAs Manager, it immediately tries to reserve its original path as specified in the static allocation plan. This original path, consisting of an ordered list of nodes to reach, together with the arrival and departure time for each of them, becomes the UA's "current path". To reserve a path, the UA sends one request to each RA in charge for the resources labeling the nodes in the path. These requests are stored in the UA's knowledge base with "waiting" state and remain valid only for a specified amount of time ("time out"). The UA waits for the answers from the RAs only for this amount of time. The requests with an elapsed time out change their state from "waiting" to "not accepted".

```
// ------------------------------------------------------------------------------
//                       definition of function void main()
//------------------------------------------------------------------------------

void main()

\\ the current path is initialized with the original path as stated in the static allocation plan;
\\ we do not describe here how/where the original path is stored
currentPath = retrieveOriginalPath();
sentRequests = {}; \\ the set of sent requests is initially empty
acceptedAnswers = {}; \\ the set of answers accepted by Resource Agents is initially empty
refusedAnswers = {}; \\ the set of answers that cannot be accepted by RAs is initially empty
toBeStolenAnswers = {}; \\ the set of requests that the agent can steal is initially empty

for each Step s in currentPath
begin
   \\ create a new message with the information regarding the reservation request,
   \\ that are: the RA, arrival and departure time, information about the User agent
   message m = createRequest(s);
   send(QUERY_IF, m.RA, m); // a request is sent to the RA in charge for the node
   sentRequests = sentRequests U request(m); // the request is stored with WAITING status

   \\ the agent will start a countdown for each request: at the end of this predefined timeout,
   \\ it will check the status of its requested reservation calling the TimeOutElapsed().
   \\ If the answer for a reservation arrives before the TimeOut ends, the message is managed with the
   \\ receivedMessage(message) procedure and the countdown for the request is deleted.
   startTimeOut(m);
end

sleep(); \\ the agent will wait until a new message is received or one of the TimeOuts ends

end
```

```
// ------------------------------------------------------------------------------
//                      definition of function void TimeOutElapsed()
//------------------------------------------------------------------------------

void TimeOutElapsed() \\ this procedure automatically starts when a TimeOut t ends
  message m = t.getMessage();
  sentRequests = sentRequests - request(m); \\delete the current request from the list of those waiting
  refusedAnswers = refusedAnswers U refuse(m); \\add the waiting request to the refused requests
  checkPath(); \\ decide what to do: see later on the details
end
```

The UA processes all the answers it receives until:

1. All the expected answers arrived before the time out expiration and they are all “ok”. In this case, the UA confirms all the reservation requests.
2. All the answers arrived before the time out expiration but at least one of them was marked as “not accepted” (namely, it was either “occupied” or “not available”). The UA looks for another solution, as discussed in the sequel.
3. The time out elapsed before all the expected answers arrived. The situation is similar to the previous case: reservation requests that were not served in time are marked as “not accepted” and the UA looks for another solution.
4. One of the answers, arrived before the time out expiration, makes the delay too high (as explained later on): the UA looks for another solution, as discussed in the sequel.

```
// ------------------------------------------------------------------------------
//             definition of function void receivedMessage(message m)
//------------------------------------------------------------------------------

\\ this procedure is automatically called when a sleeping agent receives a new message
void receivedMessage(message m)

  \\delete the previous request from the list of those waiting
  sentRequests = sentRequests - request(m);

  if(m.performative == CONFIRM) then
    acceptedAnswers = acceptedAnswers U accept(m); \\ add m to the accepted requests
  else if(m.performative == INFORM) then
    refusedAnswers = refusedAnswers U refuse(m); \\ add m to the refused requests
    UpdateCurrentDelay(m); \\ this procedure updates the value of the total current delay
  end
  end

  checkPath(); \\ the agent checks if all the requests have been answered and decides how to proceed
end
```

```
// ------------------------------------------------------------------------------
//                    definition of function void checkPath()
//------------------------------------------------------------------------------

void checkPath()

if(sentRequests is empty) then
  stopTimeOut(); \\ every RA answered, there is no need to wait longer

  if(allRequestAccepted()) then \\ all the reservation requests have been accepted
    confirmPath(currentPath); \\ confirm all the request reservations
    pathOk = true; \\ global variable that states that the path is correctly reserved
    sleep(); \\ if no problems arise, the UA has finished its work, it has a reserved path
  else
    checkStatus(m); \\ decide what to do considering the current answers and total delay
  end

  else
    checkStatus(m);
  end

end
```

Let us describe the behavior of a User Agent *UA1* (that is the same behavior of any other UA). *UA1* has a private knowledge of the maximum delay that it can undergo. The "occupied"/"not available" answers from RAs include the time instant when the resource will be free again. Using this information *UA1* can compute the delay it is undergoing (adding the expected delays as indicated in the RA answers) and it can decide if it can change only the arrival/departure time of nodes reservations or if it has to find another path in the graph, to get less total delay.

When an answer from an RA is received (and is not "ok") *UA1* computes the delay it would gain by maintaining the current path (that is, maintaining that RA) and delaying the arrival and departure times, and behaves according to the following rules:

- The computed delay is acceptable: the path is maintained, *UA1* waits for all the answers from RAs and when they have arrived it changes the arrival and departure times using the information in the RAs' answers. Then it sends new requests to the RAs with new time slots, re-starting the process of reserving a path (which is the same path as before, but with different arrival and departure times in the nodes).

- The computed delay is not acceptable: *UA1* acts in two different ways with respect to its priority. When a RA answers to a request with an "occupied" message, it also includes the priority of the UA that holds the reservation (let us suppose that it is *UA2*).

    1. If the priority of *UA1* (the agent sending the request) is equal or greater than the priority of *UA2* currently holding the resource, *UA1* can steal the reservation to *UA2*. *UA1* keeps waiting for the other answers but saves the information that it can stole that resource. When all the answers arrive, it confirms the reservations and, for the RA

that it wants to stole, specifies that it won the implicit contest with *UA2*, succeeding in getting the resource.

2. If the priority of *UA1* is lower than the one of *UA2*, *UA1* cannot steal the resource and must look for another path. In this situation *UA1* will stop waiting for the other answers and will look for another path, asking to PA the list of possible alternative paths:

   (a) If the list of paths returned by the PA is not empty, *UA1* selects the first path in the list, calculates the arrival and departure times to/from each node in the path[1] and sends the reservation requests for this new path, re-starting the reservation process. Once obtained all the answers from the RAs, *UA1* decides whether to confirm the reservation or not:

   - the path is acceptable (either all the reservations were successful, or some of the reservations must be delayed because the nodes are not free in the requested time slot, but they can be reserved later without causing too much delay, or some resource can be stolen): the requests are confirmed or changed with the new time slots (if some delay must be taken into account) and sent again; when the RA will send a confirmation of their availability, *UA1* will confirm their reservation;
   - the path is not acceptable (too delay or some node out of order or not responding or no resource can be stolen). *UA1* records the delay associated with this unacceptable path, and tries to reserve the next path returned by the Path Agent. This process can be iterated for P times, where P is a configurable threshold. If, after iterating the reservation for all the first P paths (or for all the returned paths, if they are less than P), no path turns out to give an acceptable delay, *UA1* acts as shown in b).

   (b) If all the P paths in the list are not acceptable because they create too much delay, *UA1* reserves anyway the one with the lowest delay.

   (c) If the list of paths returned by the PA is empty or all the P paths in the list have a node not responding, *UA1* tries again to reserve the path it was considering before querying PA, but will act in a different way considering the state of the original path:

   - If the original path was "blocked" (some node did not answer at all) it sends a Non Disputable reservation (for a predefined time slot that is longer than the original reservation) to the first node of the path and it suspends the path searching. This is the only action that can be done because the agent knows that the path has some problem, so it will wait some time and, before the

[1] In order to calculate the arrival and departure times to/from each node, *UA1* assumes that it will remain X time slots in each node and that traversing an arc requires Y time slots; *UA1* might also know that there is one special "stop node" in the path where it must stop for Z time slots.

reservation for the first node expires, it will start again trying to move on, hoping that the problem has been solved.

– If the original path created too much delay the agent will anyway reserve it accepting the delay.

```
// ------------------------------------------------------------------------------
//                   definition of function void checkStatus(message m)
//------------------------------------------------------------------------------

void checkStatus(message m)

if(CurrentTotalDelay <= MaxDelay) then

  \\ if all the answers have been received, but some of them are "occupied/not available",
  \\ update the departure and arrival times and try again to reserve the path
  if(sentRequests is empty) then
    \\ update the departure and arrival times considering the information sent by the RAs
    updateCurrentPath();
    clearLists(); \\ make all the reservation lists empty, to start again the reservation procedure
    reservePath(currentPath); \\ send the new reservation requests and wait for the answers
  else
     sleep(); \\ the agent must wait for the other answers
  end

else if(CurrentTotalDelay > MaxDelay) then

  if(myPriority > m.OtherRAPriority) then
    \\ the agent moves this request to the list of those that it can steal
    toBeStolenAnswers = toBeStolenAnswers U stole(m);
    if(sentRequests is empty) then
     \\ this procedure will check the toBeStolenAnswers list and will inform the RA that the agent
     \\ will steal the resource (and will simply confirm the requests in the acceptedAnswers list)
     confirmPath(currentPath);
     pathOk=true;
    else
     sleep();
    end
   else
     \\ the agent cannot steal the resource, so it will search for another path
     searchAnotherPath();
  end
end
end


// ------------------------------------------------------------------------------
//                   definition of function void searchAnotherPath()
//------------------------------------------------------------------------------

void searchAnotherPath();

  \\ the agent will send a request to the PA, then it will block until it receives an answer.
  \\ This function returns the list of alternative paths sent by the PA
  alternativePaths = askPA();

  if(alternativePaths is empty) then
    if(PathIsBlocked(currentPath)) then
      stopHere(); \\ let's send a Non Disputable request for the first node, and let's wait some time
```

```
      sleep(DefaultTime); \\ this procedure will suspend the agent (exiting from the current procedure)
    else
      \\ in this case, the agent will accept anyway a total delay higher than its maximum delay
      reservePath(currentPath);
    end

  else
   \\ consider the next path and try to reserve it unless the previous path
   \\ was confirmed (pathOk == true) or all the paths have been analyzed and are blocked
   while (not(pathOk) and not(noWay)) do
     newPath = getNextPath();  // retrieve the next alternative path
     if(newPath is not null) then
       clearLists();
       reservePath(newPath);
     else \\ no more alternative paths

       \\ the agent always stores if a path is blocked or not, depending on the messages content
       if(PathIsBlocked(currentPath)) then
         stopHere(); \\ let's send a Non Disputable request for the first node, and let's wait some time
         sleep(DefaultTime);
       else
         \\ in this case, the agent will accept anyway a total delay higher than its maximum delay
         reservePath(currentPath);
       end
     end
  end

 if(not(noWay)) then  \\ noWay == false means that all the paths gave a delay too high.
   \\ the agent stores, for every alternative path, the total delay it would accept,
   \\ so it is able to select the path with the minimum delay, invoking getAlternativePathWithLessDelay.
   \\ In this case the agent will accept a total delay higher than its maximum delay.
   reservePath(getAlternativePathWithLessDelay());

 else \\ all the paths were blocked

   \\ the agent always stores if a path is blocked or not, depending on the messages content
   if(PathIsBlocked(currentPath)) then
     stopHere(); \\ let's send a Non Disputable request for the first node, and let's wait some time
     sleep(DefaultTime);
   else
     \\ in this case, the agent will accept anyway a total delay higher than its maximum delay
     reservePath(currentPath);
   end

 end
 end
end
```

Note that if the UAs are free to stole resources from others with the same priority the system may enter an oscillation, because two UAs (for example *UA1* and *UA2*) may continue to stole each other one, or more, resources. To avoid this situation we let the UA, say *UA1*, remember the last UA, say *UA2*, it stole a resource from: when *UA1* is going to steal again a reservation, it will do it if and only if this reservation is not of *UA2*. This prevents *UA1* and *UA2* to enter a loop: the protocol lets *UA2* get back its reservation if it has no other solution, but prevents *UA1* from stealing again the resource from *UA2*.

The Path Reservation procedure just described is quite complex but, simplifying and summarizing it, we can say that the UA will always try to reserve a complete path, from a starting node to an exiting one, using this priority rules:

1. It accepts to get delay and remains on the same path until the total delay is under a specified "Maximum value" (even if it could steal a resource) or
2. If the delay is too high and the UA priority is higher than that of other UAs already holding one, or more, resource, it steals the resources and remains on the same path or
3. If it cannot steal (and the total delay is too high), it looks for another path.
4. If and only if all paths are blocked (out of order nodes or not answering nodes), it stops searching and waits some time, then starts again

In Appendix B the reader can find the simplified code of the UA agent and can see how this behavior is implemented.

If a UA succeeds in getting the reservation for all the nodes of the path, it suspends until it receives a new message.

If a UA receives a "cancel" message from a RA, this means that it has lost a resource (another UA stole it or the resource became no longer available for that interval: the cause is specified in the RA's message). In this case the UA cancels all its confirmed reservations by sending a "cancel" message to all the RAs involved and re-starts the process of finding a path.

A particular message that a UA can receive from a RA is the one that will inform it that the node (on which the UA is) is out of order: in this situation, the RA informs the UA that it must stay on the resource since a specified time, reported in the message. In this case the UA has to cancel its current path (that is, cancel all its other confirmed reservations) and it will try to reserve the same path but shifting all the reservations with respect to the delay it got. In this way, while the UA is waiting on an out of order resource, it will try to be ready to move on, without other delay, as soon as the resource will be available again.

4.2.2.3 Life cycle and rules of a Resource Agent

RAs have two main goals in the protocol:

1. Maintain the state of the graph updated
2. Interact with UAs to help them finding a path in the graph

To fulfill the first goal the RAs need to interact with each others, as described in the next paragraph.

When a RA is created by the RAs Manager, its knowledge base is filled with information about its neighbors, about the arcs it has to manage, and about conflicting arcs. Let us suppose that RA controls resource R, where R is the label of node N in the graph. The neighbors of RA are those RAs that control resources labeling nodes N_1, ..., N_m, from which one or more arcs depart towards N. For every arc connecting to a neighbor, the RA knows the list of conflicting arcs, that is, the list of RAs that must be informed when the state of the arc is changed. It may happen that a reservation made by a UA over a resource affects another RA because it will have no free and non-conflicting arc left. Then, every change to the local reservation table must be communicated to the neighbors in order to allow RAs to have an up-to-date and synchronized view of the state of the arcs.

This synchronization is made in two steps: when a RA receives from a UA a reservation request it will chose a free arc, if any, it will answer the RA and it will inform its neighbors about this request. If the RA will also receive a confirmation of the reservation, it will send a confirmation to the interested neighbors. Otherwise, if the UA will not confirm, the previous information sent to the neighbors will be automatically canceled after a predefined time out.

Summarizing, the RAs will exchange different types of messages, to update the state of the graph:

- information about a request on an arc;
- information about a confirmation of a request on an arc;
- cancellation of a confirmed request on an arc.

Moreover, the RAs are also in charge of managing what we call "out of order status" of the resource they manage. The RAs Manager will inform RAs when the resource they control is out of order (that is, it is no longer available for the UAs) and the RAs will consequently inform the UAs interested in reserving it (as described later).

```
// ------------------------------------------------------------------------------
//                       definition of function void main()
//------------------------------------------------------------------------------

void main()

\\ the RA reads the information regarding the entering arcs.
\\ In the NeighborsList list there are information regarding
\\ every neighbor with the arcs exiting from it plus the incompatibilities
NeighborsList = retrieveNeighbors();

AcceptedRequests = {}; \\ the set of reservation request that the RA has already accepted
ConfirmedRequests = {}; \\ the set of reservation requests that the UAs have confirmed
```

```
RefusedRequests = {}; \\ the set of requests that the RA cannot accept

\\ the RA will sleep until a message from a RA or from a UA is received
sleep();

end


// ------------------------------------------------------------------------------
//                      definition of function void MessageFromRA(message m)
//------------------------------------------------------------------------------

\\ when a message is received, this function is called if the sender is a RA:
\\ this procedure updates the reservations of the arcs, and is based on the information
\\ coming from the Neighbors;

\\ A Reservation keeps all the information regarding a resource reservation: the UA,
\\ the starting and ending time, the used arc, the RA the arc origins from
Reservation r = getContent(m);

\\ retrieve the correct Neighbor from the NeighborsList
Node fromNode = getNeighbors(NeighborsList,m.sender);

\\ update the status of the arc specified in the message
if (m.performative == INFORM) then

  addRequest(Node, r);

else if (m.performative == CONFIRM) then

  confirmRequest(Node, r);

else if (m.performative == CANCEL) then

  cancelRequest(Node,r);

end
end
end
```

To fulfill the second goal, namely that of helping UAs to find a path in the graph, the RAs need to interact with UAs, as described in the next paragraph.

The standard interaction among a RA and a UA regards the reservation of a resource, and it is organized as a Three-way Hand Shaking:

- UA sends a request to the RA;
- RA answers to the UA;
- UA confirms its request / stoles the resource from the previous UA or
- UA does not confirm/stole and interrupts the Three-way Hand Shaking.

RAs receive requests from UAs containing the following information:

- node from which the UA arrives,
- arrival and departure time,
- priority of the UA,
- information about the disputableness of the request.

Every RA answers according to the rules below:

- if the resource is not yet occupied and there is one free entering arc A, it answers "ok": then it will update its reservation table (memorizing that there is a pending reservation on arc A) and informs of this pending request the interested neighbors;
- if the resource is not yet occupied but there are no free arcs for reaching the resource in the time slot specified by the UA, it looks for the first time instant when an arc will become available in its reservation table; it answers with a "not available" message including the first time instant when the resource will become free again;
- if the resource is already occupied by another UA but this allocation is "Disputable", it sends:
  - an "occupied" message specifying when the resource will become free again and the priority of the UA that currently holds it if the received request is Disputable;
  - an "ok" message if the received request is Non Disputable;
- if the resource is already occupied by another UA and the allocation is "Not Disputable", it answers with a "not available" message including the time instant when the resource will become free again;
- if the resource is out of order, it answers with a "not available" message including the time instant when the resource will be fixed (when the "out of order status" will end).

Every request received by the RA has a time out (that is the time interval when the RA has a priority on the resource and it can confirm its reservation). If the resource is free and there are other requests from other UAs that are still valid, the RA waits for possible confirmations from them: after the time out, it will consider the new request from the UA and, if it is able to find a free entering arc, it will answer "ok" to it.

Otherwise, if during the time out period one of the "pending" UAs confirms its request, the RA immediately answers "occupied" to all other UAs that tried to reserve the resource. In this way,

if two UAs try to reserve the same resource for the same time slot, the first that asks is the one that wins.

After the RA has answered a request as shown before, the RA will wait for other messages from UA (to conclude the Three-way Hand Shaking). It can receive a confirmation from the RA or a "stealing confirmation", that is an information from the UA that it decided to stole the resource from another UA. In this case the information in the message is the same as in the "confirm" one, and the RA will accept this confirmation but will also inform the previous UA that it has lost its reservation. So in this situation the RA has no active role: it is only in charge of informing the UA that its reservation has been stolen by another UA (that will be specified in the message). The role of the RA is passive because all the UAs will act with respect to their priority, so no further controls have to be done on the reservations.

```
// ------------------------------------------------------------------------
//                      definition of function void RequestFromUA()
//------------------------------------------------------------------------

\\ when a message is received, this function is called if the sender is a UA and it is the first step
\\ of the Three-way Hand Shaking (reservation request):
\\ this procedure updates the reservation table of the resource and informs the interested Neighbors;

\\ A Reservation request keeps all the information regarding a resource reservation:
\\ the UA and its priority, the starting and ending time, the type of the reservation
\\ (disputable or not), the RA the UA is coming from
Reservation NewRes = getContent(m);

if(free(NewRes.TimeInterval)) then
  \\ this function will return the arc to be used to reach the resource, if a free arc exists,
  \\ otherwise it will return null and will fill the global variable NewInterval
  \\ with the first free TimeInterval.
  Arc a = getFreeArc(NewRes.TimeInterval);

  if(a is not null) then
    \\ inform the UA that the resource is free and that it can use arc a to reach it
    sendMessage(CONFIRM, NewRes.UA, a, "ok");
    \\ add this request to those waiting for a confirm from UA
    AcceptedRequests = AcceptedRequests U NewRes;

  else
    \\ inform the UA that the resource will be available only in NewInterval,
    \\ because there are no free arcs
    sendMessage(INFORM, NewRes.UA, NewInterval, "NoArc");
    \\ add this request to those that cannot be accepted
    RefusedRequests = RefusedRequests U NewRes;

  end
else
\\ if the resource is not free, let's consider the type of the two reservations
Reservation OldRes = findReservation(NewRes.TimeInterval);

if(isDisputable(NewRes) and isDisputable(OldRes)) then
    \\ inform the UA that the resource is already reserved by the UA that made OldRes
    sendMessage(INFORM, NewRes.UA, NewInterval, OldRes.UA);
    RefusedRequests = RefusedRequests U NewRes; \\ add NewRes to those that cannot be accepted
else if(isDisputable(OldRes) and not(isDisputable(NewRes))) then
```

```
    \\ this reservation must be accepted
    sendMessage(CONFIRM, NewRes.UA, OldRes.arc, "ok");
    AcceptedRequests = AcceptedRequests U NewRes; \\ add NewRes to those waiting for a confirm

else if(not(isDisputable(OldRes))) then
    \\ OldRes cannot be stolen because it is not disputable, so the RA specifies a false UA, with
    \\ very hight priority: in this way no UA can steal OldRes
    sendMessage(INFORM, NewRes.UA, NewInterval, FalseUA);
    RefusedRequests = RefusedRequests U NewRes; \\ add NewRes to those that cannot be accepted

end
end

end

\\ inform the Neighbors
if(a is not null) then
  \\ retrieve the sublist of Neighbors, from NeighborsList,
  \\ managing an arc with incompatibilities with a
  NodeList nodes= getNeighbors(NeighborsList , a);

  for (each Node n in NodeList) sendMessage(INFORM, n, NewInterval, a);

end
end


// ---------------------------------------------------------------------------
//                      definition of function void ConfirmFromUA()
//----------------------------------------------------------------------------

\\ when a message is received, this function is called if the sender is a UA and it is the
\\ last step of the Three-way Hand Shaking (confirm of a reservation request): this procedure
\\ updates the reservation table of the resource and informs the interested Neighbors;

\\ A Reservation request keeps all the information regarding a resource reservation:
\\ the UA and its priority, the starting and ending time, the type of the reservation
\\ (disputable or not), the RA the UA is coming from
Reservation NewRes = getContent(m);

\\ move the reservation from those waiting/refused to those accepted
AcceptedRequests = AcceptedRequests - m;
RefusedRequests = RefusedRequests - m;
\\ note that the RA will check both lists because it does not know if the UA
\\ is confirming a reservation or it is stealing a reservation

ConfirmedRequests = ConfirmedRequests U m;

\\ check if the were other Disputable reservations that must be canceled
\\ or if the UA is stealing a reservation from another UA
informOtherUAs(m);

\\ inform the Neighbors
if(NewRes.Arc is not null) then
  \\ retrieve the sublist of Neighbors, from NeighborsList,
  \\ managing an arc with incompatibilities with NewRes.Arc
  NodeList nodes= getNeighbors(NeighborsList, NewRes.Arc);

  for (each Node n in NodeList) send(INFORM, n, NewRes.TimeInterval, m);

end
end
```

Lastly, when a RA is informed by the RAs Manager that its resource is out of order, it must manage this update: this information is sent when the unavailability starts, specifying also when the resource will be available again (we call this time T in the sequel). The RA will act as described:

- Check if in that moment a UA, let it be *UA1*, is already on it and if it should leave the node before T: if true, it will inform *UA1* that it must wait on the resource until it is again available (after T)
- Check if other UAs, after *UA1*, had already confirmed requests for the period when the resource is out of order: if true, inform them that the node is no longer available (send them a "cancel" message)
- Answer UAs which need a reservation during the out of order period that the resource is "not available"

4.2.2.4 Convergence towards a solution

According to the features of the real application that we addressed, the negotiation algorithm converges towards a solution provided that there are always enough free nodes for allowing each UA to cross the graph even if following a path different from the original one. The number of arcs in the graph ensures a redundancy in the choice of paths (most of the nodes are reachable from at least two nodes, and at least two arcs depart from them). Also, UAs enter the graph at the time stated by their static allocation plan or later.

The static allocation plan itself ensures that UAs enter the MAS at a "reasonable" pace. This avoids that too many UAs are in the graph at the same time instant, generating a large amount of re-negotiations that might cause some UA to undergo continuous plan changes.

Although in the real application for which the negotiation algorithm has been designed it is very unlikely that two UAs reserve the same node with "non disputable" priority, the situation may nevertheless occur. It represents a final solution too, in the sense that the MAS realizes that it cannot cope with it because, considering the current parameters (for example the maximum acceptable delay), it is impossible to find a new position in the station for all the UAs. In our real application, if two UAs attempt to reserve the same node with "non disputable" priority they evidence a serious problem: the number of damaged nodes and arcs in the graph is far above the physiological number of failures, and this crisis must be faced by a human expert.

Anyway, this situation will never be solved by FYPA violating the constraints: if two UAs need to stop on a resource using a "non disputable" reservation, only the first will succeed in getting

it. The second UA will have to find another solution, that is to stop, with a "non disputable" reservation, on a previous node. This situation will lead to make some UAs waiting out of the station (that is, to enter later). This solution is acceptable for FYPA because it does not violate any constraints but it is a solution that must be checked by an human, because if in FYPA the railway out of the station is seen as an infinite railway, it is not physically infinite and it is not a good solution to have many trains waiting outside the station. Probably, this situation will be solved stopping some trains in the previous station, but this is out of the FYPA system managing and out of the aim of our study.

4.2.2.5 Does FYPA solve the original MARA problem?

Considering the original Ansaldo STS MARA problem that the FYPA protocol had to solve

> *For each entity that enters the graph from a start point either confirm the validity of the plan stated in the static allocation plan or, if some unexpected event occurred that makes the original plan no longer applicable, find a new consisting plan for reaching the end point of the graph as stated in the original plan. The new plan should get the entity exits the graph with a low delay, modifying the original plan the less is possible, as well it should involve few entities in the re-allocation process.*

we could state that we achieved this goal. Summarizing from the previous sections, we now briefly show how the different constraints of the MARA problem have been solved. In the next chapters more details will be given to show how the protocol has been implemented, while here we limit our proof to the analysis of the FYPA protocol design.

1. "The new plan should get the entity exits the graph with a low delay": every UA has a maximum delay (MD) it can accept while searching for a new path in the graph. When the UA must recalculate its path (that is, it must change the arrival and departure time for some nodes or it must change the nodes of the path), it accepts the new path if the total delay is less than the MD. Only if it is not able to find any other solution it will accept more delay than MD.

2. "...modifying the original plan the less is possible": when a UA asks the PA for the list of alternative paths, it will send also the information regarding its current path (that is the original plan when it asks the PA the first time). These information let the PA answer with an ordered list of alternative paths, that will always start with the path most similar to the current one. The UA will try to reserve a new path starting from those that are more similar to the original one, stopping when a free path is found. In this way, the new reserved path is the one (currently free) most similar to the original one.

3. "The new plan should involve few entities in the re-allocation process": when a UA asks a RA to reserve a resource, the RA will answer only after that the UAs which previously requested the resource have answered. In this way, the protocol avoids the UAs repeatedly steal the resource because the RAs already implicitly suggest to the UAs what to do (specifying the priority of the other UAs and informing about the current delay to reserve a resource). Moreover, the UAs are allowed to steal a resource only if they have an higher priority or if they have the same priority. In this second case, they are not allowed to enter an "oscillation" (for example, UA1 steals a resource to UA2 that "steals back" to UA1 and again UA1 steals to UA2...) but they are forced to find another solution. So the conflicts for a resource are limited to two UAs (the one that has the reservation and the first one trying to reserve it) each time: moreover, the priority is implicitly used to select (and to reduce the number of) the possible UAs to be involved (stealing them a resource).

4. "...find a new consisting plan": a RA will never accept two intersecting reservations, even if they are "non disputable", and a RA will also never allocate an arc with incompatibilities that is already reserved by another RA.

The system implementing the FYPA protocol finds a solution within two seconds (as described in the next chapters), so the constraint to find a solution within a predefined amount of time is respected.

4.3 Implementation of the MAS

The "Ansaldo FYPA" system has been implemented with JADE and exploits JADE WSIG (Web Services Integration Gateway, that is an add-on that provides support for invocation of JADE agent services from Web service clients and vice versa) for interfacing the Ansaldo STS application outside the MAS. The methods which interface the Ansaldo STS application have been developed by Riccardo Caccia.

We use the standard FIPA-ACL messages, containing a Java String as body and specifying a performative act from those predefined. So the MAS is FIPA compliant.

The system is organized in four main packages (User and Resource agents, User Agents Manager, Resource Agents Manager and Paths Manager) which contain approximately forty classes (10 classes each, more or less). More than 9000 lines of code are divided into the main five classes, managing only the agents behaviors, and other 3000 lines are spread among the classes which represent the complex objects and lists used by the agents. These five classes will be analyzed in details later on. In our tally we only include the lines of code developed by the author of this Thesis. Other pieces of code have been developed by the scientist in Ansaldo STS, but of course we did not count them.

In the next sections we will describe the implementation of Resource and User Agents in the "Ansaldo FYPA" system. Note that in the next sections we will call:

- the User Agent "Treno"
- the Resource Agent "Nodo"
- the User Agents Manager "MovimentiIndotti"
- the Resource Agents Manager "PrenotazioniStazione"
- the Paths Manager "PercorsiAlternativi"

reflecting the name of classes and entities in the real application.

4.3.1 Implementation of Resource Agents (Nodo)

Any Resource Agent is an instance of the class $Nodo.java$, that extends $(JADE)Agent$.

The core of the agent is a $CyclicBehavior$ where Nodo waits for new messages.

All messages from PrenotazioniStazione and Nodo agents are immediately parsed, "used" (that is, they make the agent immediately perform an action) and canceled. Otherwise, messages from Treno agents are usually not only immediately parsed but they are stored in different lists (that is, the Nodo stores the information contained in the message in specified objects and lists) and the agent will use them later, in other phases of the protocol.

4.3.1.1 Messages among Nodo agents

In Table 1 a list of possible messages received by a Nodo agent (NA) is reported: the first column shows the performative type, in the second column there is a description of the message and in the last column the action to be taken under reception of the message is described. These messages allow Nodo agents manage the state of the entering arcs, so they deal only the reservation of arcs, not of resources. Moreover, we assume that all the arcs have a "traveling time", so when a resource is reserved from time $T1$ to $T2$ and arc A is used to reach the resource, the arc will be occupied only from $T1$ to *T1 + "traveling time"*.

Definition of "neighbors" of a Nodo NA. Let *[A1 from N1, A2 from N2..., AN from NN]* be the list of entering arcs in NA: the "neighbors" of NA are those agents from which an arc, entering in NA, originates (i.e., $N1$, $N2$, ...NN) and those agents managing an entering arc which has an incompatibility with $A1$ or $A2$... or AN.

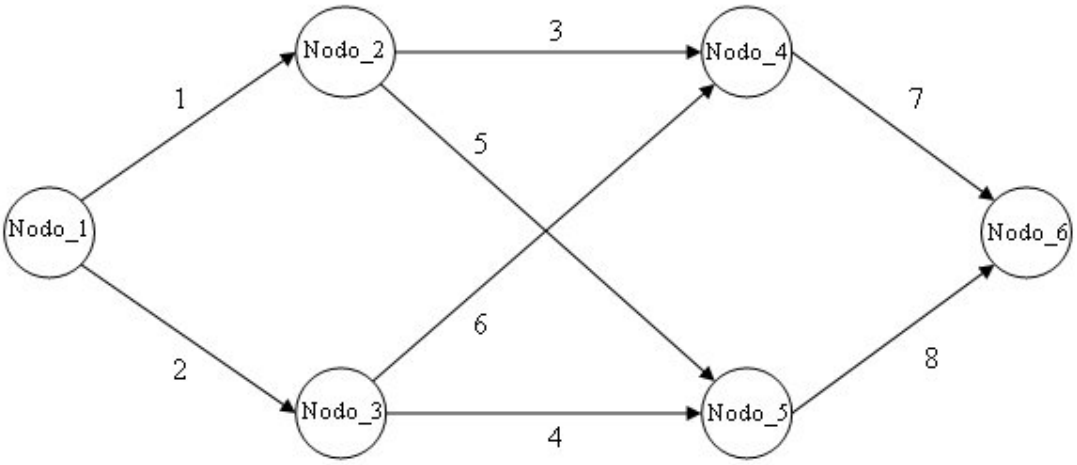


Figure 1: Graph's Example

For example, in Figure 1 the "neighbors" of $Nodo\_4$ are $Nodo\_2$ and $Nodo\_3$ but also $Nodo\_5$, because of the incompatibility between arcs number five and six.

The neighborhood relationship is not symmetric, unless the arc is bidirectional. For example, in the previous graph $Nodo\_2$ is a neighbor of $Nodo\_4$ but $Nodo\_4$ s not a neighbor of $Nodo\_2$.

Agents can send messages only to their neighbors, but can receive messages also from agents that are not their neighbors.

For every performative, the content of the message received from a Nodo agent has always this format:

[(arc1,nodeA),(arc2,nodeB)...];From;To;Conf;Train;inf.

- *[(arc1,nodeA),(arc2,nodeB)...]*: List of couples (Arc Identification Number, Origin Node). Nodo agent will use it to find which entering arc are interested by this message and from which node they origin
- From: Starting Time of the reservation. It is the same value of the reservation of the train
- To: Ending Time of the reservation. This value is reported but it is not used: the Ending time of the reservation will be equal to the Starting Time value plus the predefined "traveling time", for every arc (it is a built in time constant, usually few seconds).
- Conf: State of reservation (requested/confirmed).
- Train: Train that made the reservation. It refers to the name of the Train that made the original request for the resource reservation

- inf: Used for internal control and testing

Note that Table 1 reports messages that can be received by a Nodo agent NA. They are the same that the NA will send to its neighbors, but in the table the point of view is the one of the receiver.

Performative	**Description**	**Action**
INFORM	This message is received by Nodo agent NA from agent E to inform it about a reservation request made by a Train for a specified time using an arc A. NA is informed because arc A is bidirectional or because it has incompatibilities with arcs *arc1, arc2...*, entering in NA. In this type of message the value of field "conf" will be false. Note that arc A is reported only if it is bidirectional (the list reports the value (A, E)), otherwise it does not matter.	Nodo Agent NA will search its lists for arcs *arc1, arc2...* updating their status to "requested" in time interval *From* to *From* + *"traveling time"*.
CONFIRM	This message is received by Nodo agent NA from agent E to inform it that a reservation request made by a Train for a specified time using an arc A has been confirmed. This message is the "conclusion" of the reservation protocol. Its content is the same as the one of the INFORM message but the value of field "conf" will be true.	Nodo Agent NA will search its lists for arcs *arc1, arc2...* updating their status to "reserved" in time interval *From* to *From* + *"traveling time"*.
AGREE	This message is sent by a Nodo agent E to inform its neighbors (let call them NA, $N1$, $N2$ etc.) that a Train has stolen the resource in a specified period. Its content is the same as the one of the CONFIRM message.	As for the CONFIRM message, Nodo Agent NA will search its lists for arcs *arc1, arc2...* updating their status to "reserved" in time interval *From* to *From* + *"traveling time"*.
CANCEL	This message is received by an agent to inform it that an already confirmed reservation has been canceled. In this case the value of the field "conf" does not matter.	Nodo Agent NA will search its lists for arcs *arc1, arc2...* updating their status to "free" in time interval *From* to *From* + *"traveling time"*.

Table 1: List of possible messages received from a Nodo agent

4.3.1.2 Messages among Treno and Nodo agents

The Nodo agent uses four lists to maintain an updated view of its interaction with Treno agents:

- lista_prenotazioni: list of confirmed reservations
- lista_prenotazioni_rich: list of reservation requests the Nodo agent already answered to with a CONFIRM

- lista_prenotazioni_att: list of reservation requests waiting for the Nodo answer. Nodo has not answered yet because it is waiting for other Train answers. That is, when the Nodo receives, for the same time interval, more than one reservation request, it stores the first request in the list $lista\_prenotazioni\_rich$ and the others in list $lista\_prenotazioni\_att$. When the first Train answers, the Nodo agent answers to the reservation requests in this list (and the response will be INFORM, if the first train confirmed its request, or CONFIRM to the first waiting request and INFORM to the others), moving them in the other lists ($lista\_prenotazioni\_occ$, $lista\_prenotazioni\_rich$)
- lista_prenotazioni_occ: list of reservation requests that received an INFORM answer.

In Table 2 a list of messages from Treno agents is reported: in the first and in the second column the performative type and a description of the message are reported while in the last column the action to be taken by the Nodo agent on reception of this message is reported.

The content of the ACL message will always have this format:

PR#;From;To;FromNodo;Disputable;Priority

- PR#: Number of reservation request. Every Train assigns a unique number to every new request reservation. This number can be the same for different trains.
- From: Starting Time of the reservation (including the "traveling time" of the entering arc).
- To: Ending Time of the reservation.
- FromNodo: The Node from which the train will reach the current Nodo.
- Disputable: Boolean value specifying whether the reservation request is disputable or not.
- Priority: the priority of the Train.

The messages are definitely deleted from lists after a CANCEL message from a Train or when they are old, that is when the time reached by the simulation is bigger than the To value of the reservation.

In Table 3 a list of messages sent to Treno agents is reported: in the first and in the second column the performative type and a description of the message are reported, while in the last column the state that generated the message to be sent is reported.

Performative	Description	Action
QUERY_IF	This message is received by Nodo agent NA from a Train agent T which asks the NA if the resource is free in the period "From - To". The Train specifies the number of its request reservation, the type (if it is disputable or not) and its priority, so NA has all the information it needs to answer the request. This message is the first of the Three-way Hand Shaking protocol to reserve a resource.	This message will be moved in the list: • lista_prenotazioni_rich if the agent answers with a CONFIRM message • lista_prenotazioni_occ if the agent answers immediately with an INFORM message • lista_prenotazioni_att if the agent must wait for another message from another Treno agent before answering
CONFIRM	This message is received by Nodo agent NA from a Train agent T if and only if the NA answered with a CONFIRM message to the QUERY_IF message: the information regarding the reservation is reported again but the Nodo will only use the Reservation number (to identify the reservation request in the list $lista\_prenotazioni\_rich$ of the previous type messages).	This message will be removed from the list $lista\_prenotazioni\_rich$ and moved to the list $lista\_prenotazioni$. Then the Nodo agent will analyze the messages for the same time period in the list $lista\_prenotazioni\_att$ and it will move all them, if any, to list $lista\_prenotazioni\_occ$ (and the agent will answer with an INFORM message to the Trains that sent the messages just moved).
AGREE	This message is received by Nodo agent NA from a Train agent T which stoles a reservation (from Train PT). This message is sent after the train received an $INFORM$ message from the Nodo. We use the AGREE performative because Train T informs NA that T and PT agreed on the change of reservation. This agreement is implicit (PT is not really involved in the exchange) so we say that "T stole the resource".	The Nodo agent will search for the previous request from the train in the list $lista\_prenotazioni\_occ$ and will move it in the list $lista\_prenotazioni$, but before doing this it will inform the other Trains (which had a confirmed reservation) that they lost their reservation. Lastly the Nodo agent will check also the lists $lista\_prenotazioni\_att$ to inform the other waiting Train agents of this new reservation.
CANCEL	This message is sent by a Treno agent to cancel one confirmed reservation.	The Nodo agent will search in the $lista\_prenotazioni$ list the confirmed reservation and will delete it.

Table 2: List of possible messages received by a Nodo agent coming from a Treno agent

Performative	Description	Action
INFORM	This message is sent to a Treno agent T to inform it that its request reservation can not be accepted because another Train ($T2$) has a previous reservation. The content specifies the number of the reservation of T, its state (false), the interval when the Nodo will be again available and data of the other Train ($T2$). *ReservationNumber;false;NewAvailabilityStartTime; NewAvailabilityEndTime;Priority of T2; Name of T2*	This message is sent back to a Train, answering a $QUERY\_IF$ message, if the resource is not free (another Train, $T2$, has already a reservation).
CONFIRM	This message is sent to a Treno agent to inform it that its request reservation has been accepted. The content specifies the number of the reservation, its state (true) and the number of the arc that can be used to reach the Nodo. $ReservationNumber; true; Arc$	This message is sent back to a Train, to answer a $QUERY\_IF$ message, if the resource if free or has only other Disputable reservations.
FAILURE	This message is sent to a Treno agent T to inform it that it must wait more on the resource than planned (till $MustStayTo$). Values -1 and *"rottura nodo"* specify that the node is out of order. The content format is: *ReservationNumber;false;MustStayFrom; MustStayTo;-1;"rottura nodo"*	This message is sent when the Nodo agent goes out of order and T is already on it.
CANCEL	This message is sent to a Treno agent T to cancel one of its confirmed reservation ($ReservationNumber$). The data of the thief (if any) are reported too. The format is: *ReservationNumber;false;NewAvailabilityStartTime; NewAvailabilityEndTime;Priority of thief; Name of thief*	This message is sent when one other train steals the resource to T or when the Nodo agent goes out of order (but train T is not already on it) or when the Nodo agent receives from T a CONFIRM/AGREE for an expired reservation request.
PROPAGATE	This message is sent to a Treno agent T to inform it that it has to modify the arc to be used for its confirmed reservation ($ReservationNumber$). The format is: *ReservationNumber;NewArc;*	This message is sent when one arc goes out of order: if the Node is able to find another free arc FA it will substitute the out of order arc with FA and will inform the interested Trains.

Table 3: List of possible messages sent to a Treno agent

4.3.2 Implementation of User Agents (Treno)

Any User Agent is an instance of the class $Treno.java$, that extends $(JADE)Agent$.

The core of the agent is a $CyclicBehavior$ where Treno tries to reserve all the resources it needs to leave the station. So this agent will alternate an active phase, where it sends Nodo agents messages to get the reservations, and a waiting phase, where it waits for the answers from Nodo agents (or where it is simply suspended because it has already reserved a complete path).

4.3.2.1 Messages among Nodo and Treno agents

The Treno agent uses four lists to maintain an updated view of its interaction with Nodo agents:

- lista_prenotazioni_richieste: list of reservation requests sent to a Nodo agent that have not yet an answer. The reservation requests are stored in this list after the first step of the "Three-way handshaking protocol";
- lista_prenotazioni_accettate: list of reservations that have been accepted by a Nodo (those with a CONFIRM answer from a Nodo agent) but that are not yet confirmed again by the Treno agent. The reservation requests are stored in this list after the second step of the "Three-way handshaking protocol" (if the resource is available);
- lista_prenotazioni_non_accettate: list of reservation requests that received an INFORM answer from a Nodo (that is, those reservation requests that cannot be accepted by the Nodo agents). The reservation requests are stored in this list after the second step of the "Three-way handshaking protocol" (if the resource is not available);
- lista_prenotazioni_confermate: list of reservation requests that have a CONFIRM answer and that the Treno agent has already confirmed (sending back a CONFIRM message to the Nodo). These are the confirmed reservations stating that a Treno agent can move on the resource at the foreseen time. The reservation requests are stored in this list after the third step of the "Three-way handshaking protocol".

In Table 4 a list of messages sent by a Treno agents is reported: in the first and in the second column the performative type and a description of the state that generated the message are reported, while in the last column the internal actions that are executed to manage the message just sent are described.

The content of the ACL message will always have this format:

PR#;From;To;FromNodo;Disputable;Priority

- PR#: Number of reservation request. Every Treno agent assigns a unique number to every new request reservation.
- From: Starting Time of the reservation (including the "traveling time" of the entering arc).
- To: Ending Time of the reservation.
- FromNodo: The Node from which the train will reach the current Nodo.
- Disputable: Boolean value specifying whether the reservation request is disputable or not.
- Priority: the priority of the Treno agent.

The messages are definitely deleted from lists after a CANCEL message from a Nodo or when they are old, that is when the time reached by the simulation is bigger than the To value of the reservation.

In Table 5 a list of messages received by a Treno agent is reported: in the first and in the second column the performative type and a description of the message are reported, while in the last column the action to be taken is reported.

Performative	Description	Action
QUERY_IF	This message is sent by a Treno agent TA to a Nodo agent N to know if the resource is free in the period "From - To". The train specifies the number of its request reservation, the type (if it is disputable or not) and its priority, so NA has all the information it needs to answer the request. This message is the first of the Three-way Hand Shaking protocol to reserve a resource.	This message is sent when a Treno agent is trying to reserve a path: in this case it will send a reservation request for each Nodes in its path. The message will be added to the list lista_prenotazioni_richieste.
CONFIRM	This message is sent by Treno agent TA to a Nodo agent N if and only if N answered with a CONFIRM message to the previous QUERY_IF message: the information regarding the reservation is reported again but the Treno agent will only use the Reservation number (to identify the reservation request in the list $lista\_prenotazioni\_richieste$ of the previous type messages).	This message will be removed from the list $lista\_prenotazioni\_richieste$ and moved to the list $lista\_prenotazioni\_accettate$. Then the Treno agent will check if it has already received all the answers from the Nodo agents and will confirm the path if all the requests have been accepted.
AGREE	This message is sent by a Treno agent TA to a Nodo agent N to stole a reservation (already reserved by PT), that is, to inform Nodo that the Treno agent can steal the resource from PT.	This message will be removed from the list $lista\_prenotazioni\_richieste$ and moved to the list $lista\_prenotazioni\_accettate$. Then the Treno agent will check if it has already received all the answers from the Nodo agents and will confirm the path if all the requests have been accepted.
CANCEL	This message is sent by a Treno agent to cancel one confirmed reservation.	The Treno agent must start again the reservation protocol because it lost a confirmed reservation: so it cancels all its confirmed reservations and start again the process.

Table 4: List of possible messages sent by a Treno agent

Performative	Description	Action
INFORM	This message is received by a Treno agent T, after a QUERY_IF message, to inform it that its request reservation can not be accepted because another Treno agent ($T2$) has a previous reservation. The content specifies the number of the reservation of T, its state (false), the interval when the Nodo will be again available and data of the other Treno agent ($T2$). *ReservationNumber;false;NewAvailabilityStartTime; NewAvailabilityEndTime;Priority of T2; Name of T2*	If the Treno agent estimates that it needs to steal the resource, it will remove the message from the list $lista\_prenotazioni\_richieste$ and it will move it to the list $lista\_prenotazioni\_accettate$. Otherwise, it will move the message to the list $lista\_prenotazioni\_non\_accettate$. Then, it will act as described in 4.2.2.2 to move on with the path reservation.
CONFIRM	This message is received by a Treno agent to inform it that its request reservation has been accepted. The content specifies the number of the reservation, its state (true) and the number of the arc that can be used to reach the Nodo. $ReservationNumber; true; Arc$	This message is removed from $lista\_prenotazioni\_richieste$ and moved to the list $lista\_prenotazioni\_accettate$. Then, it will act as described in 4.2.2.2 to move on with the path reservation.
FAILURE	This message is received by a Treno agent T to inform it that it must wait more on the resource than planned (till $MustStayTo$). Values -1 and "$rotturanodo$" specify that the node is out of order. The content format is: *ReservationNumber;false;MustStayFrom; MustStayTo;-1;"rottura nodo"*	This message is received only when the Nodo agent goes out of order while T is already on it. In this case T will modify the message in the list $lista\_prenotazioni\_confermate$ updating the To field with the value $MustStayTo$ and will try to reserve again the remaining path with the updated arrival and departure times.
CANCEL	This message is received by a Treno agent T to cancel one of its confirmed reservation ($ReservationNumber$). The data of the thief (if any) are reported too. The format is: *ReservationNumber;false;NewAvailabilityStartTime; NewAvailabilityEndTime;ThiefPriority;ThiefName*	This message is received when one other train steals the resource to T or when the Nodo agent goes out of order (but train T is not already on it), or when the Nodo agent receives from T a CONFIRM/AGREE for an expired reservation request. In this case the Treno agent will re-calculate the times for the current path or will search for another path, depending on the delay it could get shifting this reservation. This message is removed from the list $lista\_prenotazioni\_confermate$ and move in the list $lista\_prenotazioni\_non\_accettate$
PROPAGATE	This message is received by a Treno agent T to inform it that it has to change the arc to reach the Nodo: *ReservationNumber;NewArc;*	Treno agent will update the information of the previous confirmed reservation changing the arc with the one reported in the message ($NewArc$).

Table 5: List of possible messages received by a Treno agent

4.3.2.2 Messages among Treno and Paths Manager Agent (PA)

A Treno agent can contact the Path Manager agent to get a list of alternative paths in the station. Treno will send a message with performative QUERY-IF and content of the form:

Name of the train; Current Path First Node; Current Path Last Node; Current Path Stop Node; Current Path;Node to avoid

- Name of the train: it is the name of the Treno agent;
- Current Path First Node: considering the path that the train is trying to reserve, this is the first node in the path;
- Current Path Last Node: considering the path that the train is trying to reserve, this is the last node in the path;
- Current Path Stop Node: considering the path that the train is trying to reserve, this is the "Stop node" in the path (if the path contains a "Stop node");
- Current Path: this is a list, with format *[First node, Second Node... Last Node]*, reporting the actual path (to be excluded from the returned list);
- Node to avoid: if it is not null, this is the node that the train needs to avoid. It must not be included in any of the returned paths.

The PA will answer with an INFORM message with format

[Path1, Path2...] where every Path is a list with this format

(FirstNode_Is stop node?; SecondNode_Is stop node?;...)

Chapter 5

From “Ansaldo FYPA” to “Stand-alone FYPA”

In 2008 we designed a prototype working with the Ansaldo STS system: it was able to read configuration data from Ansaldo software and to send results to it, using Web Services. This was the first FYPA system, the one we called “Ansaldo FYPA”. Then, in 2009 we decided to separate our prototype from Ansaldo system to get a “standalone software”: we called this system “Stand-alone FYPA”.

The two systems are different only in the input/output management: we changed the type source of input data and we modified the output visualization, because we needed to separate the MAS from the Ansaldo STS system. The after mentioned system tests and the execution times (Section 5.6) are the same for both systems, because in those times we did not include the “load times”, that are the only ones that have been changed.

We also added a post-execution simulator to the “Stand-alone FYPA” system: this is a NetLogo program able to read the output files of the agents and to graphically represent the moves of Trains on the graph. This add-on is described later in Section 5.5.

In this chapter we will describe the main design and code changes that have been applied to the “Ansaldo FYPA” system (including the behaviors of the Interface Agents) to get the “Stand-alone FYPA” system. A detailed description of the new NetLogo interface is also reported.

In Table 1 we report a brief summary of the differences between “Ansaldo FYPA” and “Stand-alone FYPA”.

	Ansaldo FYPA	**Stand-alone FYPA**
Operative System	Linux	Windows (XP or Vista)
Input data	Text Files	Database
Interface with data	Web Services (WSIG)	Connection to DB
Alternative Paths	Saved in an external file	Calculated at run time from db
Type of arcs	Principally One-Way	Bidirectional
Graphical Interface	Ansaldo STS program	NetLogo

Table 1: Comparison between "Ansaldo FYPA" and "Stand-alone FYPA"

5.1 The Database

"Ansaldo FYPA" was strictly connected to the Ansaldo STS system, that was the source of all configuration data, run-time information and updates. This information was exchanged using a set of Web Services offered by Ansaldo STS and connected to the Multiagent system using JADE WSIG. To separate the "Ansaldo FYPA" System from the Ansaldo STS one we needed to substitute the configuration data and to find another way to access them, possibly changing as little as possible in the system: so we decided to modify only the Manager Agents. In "Ansaldo FYPA" the configuration data were physically stored on a set of text files, that were extremely complex and that contained lot of information not concerning the Multiagent System itself: so we preferred to extract only those data relevant for our system and to set up a relational database to represent them. We used the PostGres database Server. The structure of the database has been designed in order to respect the logical representation of the domain, not simply mapping the original files to tables.

It is also necessary to remember that we are representing a physical Rail station: it has a geometric structure and specific rules to cross it. In particular it can be traversed from Est to West (let us say "EW") or vice versa ("WE"), so we can always identify the nodes that lay on the western border of the station (leftmost ones) and those that lay at the eastern one (rightmost ones). So when we are searching a path in the graph we are concretely trying to cross the station in one direction, without never turning back.

In the "Stand-alone FYPA" system we assumed that all the arcs were bidirectional, that is, all paths from West to Est are the same, reversed, as those from Est to West. With this idea every Entering Node is also and Exiting Node.

We had to choose how to represent this geometric information in the database: we decided to select a direction for every arc (in Section 5.4 we explain this assumption with more details). We assumed to choose the direction of an arc (expressed as exiting and entering nodes) as if we were traversing the graph from from left to right. With this representation we will call "Entering Node" the ones on the western border and "Exiting Node" those on the eastern one. Moreover, this is only a representation formalism that does not imply any restriction on trains direction:

a train can cross the graph from Est to West or vice versa (so it can enter the graph using an "Entering Node" or an "Exiting Node").

The main tables in the database we set up are:

- Nodi (Nodes): it stores information regarding Nodes:
  - id: the key of the table;
  - nome (name): name of the Node, usually equal to the id. It is the name used to identify the agent in the system ($Nodo\_id$) and it is shown in the example's Figures;
  - entrata (way in): this boolean value specifies if it is an "Entering Node";
  - uscita (way out): this boolean value specifies if it is an "Exiting Node";
  - sosta (stop): this boolean value specifies if it is a node where usually a Train needs to stop for more time. This information is used by the Train when it is looking for a new path, as described in Section 4.2.2.2.
- Itinerari (Arcs): it stores information regarding the arcs connecting the Nodes:
  - id_itinerario (id of the arc): it is the key of the table, and it is the number that will be shown near the arcs on Figures in the examples;
  - id_nodo_uscente (id of the "from node"): it is the name of the Node the arc originates from;
  - id_nodo_entrante (id of the "to node"): it is the name of the Node where the arc enters.
- Incompatibilità (Incompatibilities): it stores the incompatibilities between arcs:
  - id_itinerario1 (id of the arc1), id_itinerario2 (id of the arc2): these are the $id\_itinerario$ of the incompatible arcs.

  Note that the relation of incompatibility is not transitive but is symmetric.
- Treni (Trains): it stores information about Trains:
  - id_treno (id of the train): this is the key of the table, and it is also the name that will be used in the system (the name of the agent will be $Treno\_id\_treno$) and shown in Figures;
  - ritardo (delay): this value represents the delay with which the Train will enter the graph, with respect to its initial plan;
  - pr (priority): it represents the priority of the Train. The lower this number, the higher the priority;

  - direzione (direction): this is a value, added for the user, to describe the direction that the Train will follow in the graph, and can be "WE" or "EW".

  Note that we call "added for the user field" those fields that contain redundant information, that is, those fields that have been added to the database schema to simplify the data insertion for the user. For example, the field $direzione$ does not add any information (because this value can be deduced by looking at the next table) but it is simpler for the user to have it because it outlines immediately the direction of the train.

- Tabelle_marcia (Original Plans): it stores information about the original plans of Train:
  - id_Treno (id of the train): it refers to the Train we are considering;
  - id_Nodo (id of the node): it specifies which Node the Train wants to move on;
  - tappa (step): it reports, for the user, the order of this step (every move on a Node is a step) in the path;
  - ora_arrivo (arrival time): it is the time, in milliseconds, when the Train will leave the previous Node to reach the specified one, or in other words, it is the time when the Train will arrive on the Resource agent minus the predefined time to pass through the entering arc, if any;
  - ora_partenza (departure time): it is the time, in milliseconds, when the Train will leave the Node.

  Note that a path is correct if and only if for each of its steps, ordered starting from $tappa$=1, the value of $ora\_partenza$ is equal to the value of ora_arrivo of the next step.

- Rotture_nodi (Out Of Order Nodes): it stores when Nodes, or arcs, will become unavailable (out of order):
  - id_nodo (id of the node): it specifies the Node we are considering;
  - da (from): it is the time, in milliseconds, when the Node will become unavailable;
  - a (to): it is the time, in milliseconds, when the Node will become again available;
  - rotto_anche_nodo (also the node is out of order): if it is true then both the Node and the arc are not available;
  - itinerario (arc): it reports the identification number of the arc that is not available. If it is equal to -1 it denotes that only the Node is out of order;
  - da_nodo (id of the "from node"): if the $itinerario$ value is different from -1, it must report the identification number of the Node the unavailable arc originates from.

We decided that only Manager Agents could access the database, with respect to their role in the MAS. So, the Resource Agents Manager is able to read tables $Nodi$, $Itinerari$, *Incompatibilità* and $Rotture\_nodi$, the User Agents Manager will access tables $Treni$ and $Tabelle\_marcia$, the Paths Manager will read from $Nodi$ and $Itinerari$.

These agents will load data only during the setup procedure, will organize this information to recreate the original "Ansaldo FYPA" structures/objects and then will use them during the simulation, without accessing the database anymore.

During the simulation every agent will use the time to understand, for example, if a reservation request is still valid or where it is in the graph with respect to its plan. Since agents use time, they must have a common perception of it and, in particular, they must share the same "Zero Time": for example, if we read from the database that a train will enter the station at time X, this X will be the number of milliseconds after the "Zero Time", and X must represent the same moment in time for all the agents in the MAS. As we know, achieving this goal in concurrent/distributed system usually is not so easy because only the PC Time is the same for every agent. Unfortunately we can not use this time in the database because in this way we could not execute the simulation every time we need (because otherwise time in tables should not be any longer relative to a generic "Zero Time" but it should be absolute).

So when the simulation starts we must find a "Zero Time": the User Agents Manager and the Resource Agents Manager are created at the same time (as far as a standard PC with one CPU can manage two threads) so when they start they will store in a variable the current PC Time (that will be the same with few milliseconds of difference) and this value will be their "Zero Time". Then when they will create Nodes and Trains they will include, as starting parameter, their "Zero Time", making it shared among all the agents in the MAS.

We know that with this solution Trains and Nodes will have few milliseconds of difference in their "Zero Time" but we verified during simulations that it does not make any difference in the results, because all the simulations already suffer from delay in threads running, Behavior execution and so on, as is normal in JADE. If the system will be enhanced we will anyway change this set up by adding a "Creator Agent" that will decide the unique "Zero Time" and then will create the User Agents Manager and the Resource Agents Manager, so that the "Zero Time" will be really the same for all the agents in the MAS.

5.2 The User Agents Manager

This agent manages the creation of Trains during the simulation. In the "Ansaldo FYPA" system an external Web service sent messages to it to create a new Train, with all the information regarding it, in real-time. In the "Stand-alone FYPA" we have no real-time source, so we have to simulate it.

The User Agents Manager in "Stand-alone FYPA" system reads, at its setup, the complete list of Trains and stores this information in a private list ordered by "Entering time" (that is the value of field ora_arrivo from table $Tabelle\_marcia$ with $tappa$ equals to 1, plus the value of field $ritardo$ from table $Treni$).

Then, it creates a TickerBehavior that every $Delta$ / 2 (where $Delta$ is predefined at compile time) looks in the list for Trains that will enter the station within $Delta$, creates them and deletes them from its list (in this way every Train is created only once). In this way Trains will be created every $Delta$ / 2 milliseconds, respecting their real "Entering time" (including the delay). Even in the "Ansaldo FYPA" system, when we say that a Train will enter the graph we are saying that the agent is created some time before its first step in the graph: in this situation we assume that the agent is waiting to enter the graph (physically, the train is waiting on a railroad out of the station) and is trying to get the reservations for its original plan. Only when the current time is equal to the time of the first reservation we can correctly say that the Train is in the station.

Summarizing, the User Agents Manager will create Trains every $Delta$ / 2, but they really will enter the graph only when the time arrives for their first confirmed reservation.

The simplified JADE code of the User Agents Manager setup is reported in Appendix C, while later on we report the pseudo-code of the main methods used by the agent.

```
// -----------------------------------------------------------------------------
//                       definition of function void main()
//------------------------------------------------------------------------------

void main(){

    // this is the Zero Time, that is the System time when the simulation is started.
    // It must be shared among all trains (and resource agents)
    ZeroTime=currentTimeMillis();

    // this procedure reads from the db the data regarding all the foreseen trains, including the delay,
    // and creates an ordered list (trains_list) considering the time when trains will enter the station
    get_train_list();

    // check if there are already trains to be created, looking in trains_list
    create_trains();

    // this procedure suspends the agent (exiting from the current procedure) for Delta/2 milliseconds.
    // Delta is a predefined value.
    // When the agent will resume, it will automatically call the function create_trains();
    sleep(Delta/2,create_trains());

  }
```

```
// ------------------------------------------------------------------------------
//                        definition of function void create_trains()
//------------------------------------------------------------------------------

// this procedure is called every Delta/2 and will create a new Treno agent
// for all those trains that will enter the station within Delta

void create_trains(){

    // class Train is used to keep the information that a Train agent will need
    Train train;

    for(int n = 0; n < trains_list.size(); n++) do
        train = trains_list[n];

        // if the train will enter the graph within Delta let's create it
        if(train.ArrivalTime < (currentTimeMillis() - ZeroTime + Delta) ) then
            // this train is removed from the list, so it will be created only once
            trains_list.remove(n);
            n--;

            // parameters for the agent to be created
            Object args[] = new Object[2];

            // this procedure reads from the database the original path of the train agent
            // and stores these information inside the train
            create_original_path(train);

            // the first parameter is the object with all the information of the train
            args[0] = train ;
            // the second parameter is the Zero Time of the simulation
            args[1]= ZeroTime;

            // a new Treno agent will be created: its name is "Treno_" plus the identification
            // number read from the db. In args the ZeroTime and all the information about
            // the Train are reported.
            createNewAgent("Treno_" + train.Id, args);

        end
    end do

}
```

5.3 The Resource Agents Manager

This agent manages the graph (the station), in other words, Nodes and arcs. In the "Ansaldo FYPA" system an external Web Service sent messages to it describing the graph and then, in real-time, informed it about new unavailabilities of some Nodes or arcs.

To simulate this organization, the Resource Agents Manager during its setup reads from the database all the information it needs and fills up the structures regarding Nodes, their entering arcs and the incompatibilities.

Moreover, to simulate the real-time notification of unavailabilities, it reads the information from the table $Rotture\_nodi$ and stores them in a list ordered by the value of the field da. Then it adds a WakerBehavior that will start after a timer equal to the first da. When this thread will be executed, the User Agents Manager will send a notification to the interested Node, will cancel this unavailability from its list and will add a new WakerBehavior for the next one, and so on. In this way Nodes are informed of their out of order status in the same way as if the system were running in real-time.

In Appendix D we report the simplified code of the Resource Agents Manager, while later on we present the pseudo-code of its main procedures.

```
//------------------------------------------------------------------------------
//                       definition of function void main()
//------------------------------------------------------------------------------

  protected void main(){
    // this is the Zero time, a global variable
    ZeroTime=currentTimeMillis();

    // this procedure will read from the db the list of the foreseen unavailabilities
    read_out_of_orders();

    // this procedure creates all the Nodo agents
    setupStation();
  }

//------------------------------------------------------------------------------
//                     definition of procedure read_out_of_orders()
//------------------------------------------------------------------------------

  private void read_out_of_orders(){

    ResultSet srs = executeQuery(
    "SELECT id_nodo, da, a, rotto_anche_nodo, itinerario, da_nodo " +
    " FROM rotture_nodi order by da");

    out_list= new ArrayList();

    // save in this list all the rows from database
    while (srs.next()) do
        // this class represents a row from the table "rotture_nodi"
        Rottura r;
        r= new Rottura(srs);
        out_list.add(r);
    end do

    if(out_list.size()>0) then
        // calculate, with respect to the Zero time, when the first "out of order status" will start
        long next_Out= out_list[0].getFrom() - currentTimeMillis - ZeroTime;

        // the agent will suspend and will resume after next_Out millisecond, that is, when the next
        // foreseen out-of-order status must be fired. When it will resume, the agent will call
        // the procedure create_out() that will inform the Nodo agents of their out of order status
        sleep(next_Out, create_out());
    end
}
```

```
//-------------------------------------------------------------------------------
//                       definition of procedure create_out()
//-------------------------------------------------------------------------------

private void create_out(){
    // this class represents a row from the table "rotture_nodi"
    Rottura r;
    long t = currentTimeMillis() - ZeroTime;

    for(int i=0; i<out_list.size(); i++) do
        r = out_list[i];

        // We must consider the execution time,
        // so we cannot check the exact value r.getDa()==t but we must
        // add some millisecond, here called T
        if(r.getDa() <= t && r.getDa() <= t + T) then

            // send a message to the Nodo agent to inform it that it is out of order
            ACLMessage msg2 = new ACLMessage(ACLMessage.INFORM);

            // the format of the content is described in the above sections,
            // and is not reported here

            msg2.setContent(r);

            // the receiver of the message is the Node interested by the out of order status
            msg2.addReceiver(r.getNomeNodoFailure());
            send(msg2);

            // delete this item from the list
            out_list.remove(i);
        end
    end do

    // let's stop and dedice when to get up (at the next out of order status)
    if(out_list.size()>0) then
        // calculate, with respect to the Zero time, when the next "out of order status" will start
        long next_Out = out_list[0].getFrom() - currentTimeMillis - ZeroTime;

        // the agent will suspend and will resume after next_Out milliseconds, that is, when the next
        // foreseen out-of-order status must be fired. When it will resume, the agent will call
        // the procedure create_out() that will inform the Nodo agent of its out of order status
        sleep(next_Out, create_out());
    end
  }


//-------------------------------------------------------------------------------
//                       definition of procedure setupStation()
//-------------------------------------------------------------------------------

private void setupStation()
  {
    // this method reads from the database the list of Nodes
    // and inserts them in the global variable nodeList
    read_node_list();

    // this method reads from the database the list of all incompatibilities
    // and inserts them in the global list ArcsList
    readIncompatibilities();
```

```
    // Node structure will keep the basic information of a Nodo
    for each Node n in nodeList do
        Object args[] = new Object[3];
        String nodeName = "Nodo_" + n.getName();
        args[0] = n;

        // In this simplified code we do not report the complete management of arcs and
        // incompatibilities because too details should be required, and they are out of scope here.
        args[1] = ArcsList;

        // this will be the Zero Time
        args[2] = ZeroTime ;

        // create a new Nodo agent
        createNewAgent(nodeName, args);

    end do
}
```

5.4 The Paths Manager

In the "Ansaldo FYPA" system the information about possible paths in the graph was managed by a Web Services external to the MAS able to answer the Paths Manager queries about the alternative paths, provided the starting and ending nodes. In that architecture the Paths Manager simply got queries and forwarded them to the external Web Service, waiting for its answer: so it only acted as a bridge between the MAS and the Ansaldo STS external system.

For this reason in the "Stand-alone FYPA" system we needed to add a new procedure to calculate all the paths. We decided not to insert in the database (or in a file) the complete list of possible paths in the graph considering all the couples of nodes (it was a "simple solution" but not a "good solution") but we preferred to calculate these paths at run time. Every time the system starts, the Paths Manager reads tables $Nodi$ and $Itinerari$ and analyzes data using a Depth First Search (DFS) starting from the entering nodes. The agent stores all the paths it finds, not only the complete ones (that are those connecting an entering node with an exiting one) because it must be able to answer, very quickly, queries where the starting and ending points are nodes of any type. Note that the agent must store two paths every time it finds one: the first is the path just found, the second is the reversed one. In the end the list with all the paths is ordered with respect to the first node of every path.

To get an efficient DFS we did not add a loop detection because we assume as precondition that data stored in the database are consistent, that is, all the arcs have been inserted respecting a specific "traveling direction" of the graph.

Let us describe an example to clarify this. In Table 2 the list of arcs in a graph is presented. Every arc has an identification number ($id\_itinerario$) and connects two nodes (in the table called $Nodo\_id$). Figure 1 represents this graph.

As the reader can observe, the list of arcs has been reported as if the graph were traversed from $Nodo\_1$ to $Nodo\_6$. This order allows us not to check the DFS procedure, because loops can not occur. In Figure 2 the complete list of paths found by the Paths Manager is reported.

If, for example, arc number 6 was expressed in the opposite direction, that is, from $Nodo\_4$ to $Nodo\_3$, and arc number 2 was from $Nodo\_3$ to $Nodo\_1$, it is simple to see that the DFS would enter a loop.

id_itinerario	**From node**	**To node**
1	Nodo_1	Nodo_2
2	Nodo_1	Nodo_3
3	Nodo_2	Nodo_4
4	Nodo_3	Nodo_5
5	Nodo_2	Nodo_5
6	Nodo_3	Nodo_4
7	Nodo_4	Nodo_6
8	Nodo_5	Nodo_6

Table 2: List of arcs

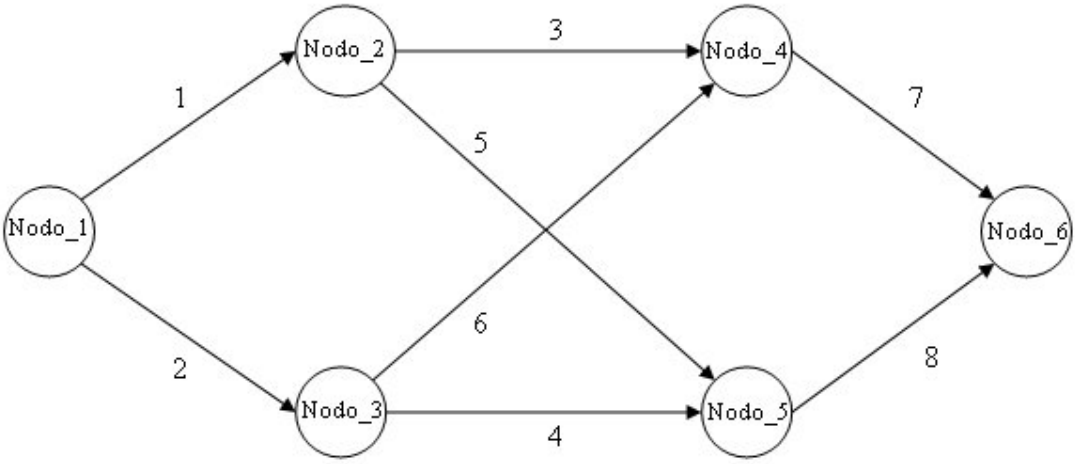


Figure 1: Graph architecture

percorso 1
nodo 1
nodo 2
percorso 2
nodo 1
nodo 2
nodo 4
percorso 3
nodo 1
nodo 2
nodo 4
nodo 6
percorso 4
nodo 1
nodo 2
nodo 5
percorso 5
nodo 1
nodo 2
nodo 5
nodo 6
percorso 6
nodo 1
nodo 3
percorso 7
nodo 1
nodo 3
nodo 4
percorso 8
nodo 1
nodo 3
nodo 4
nodo 6
percorso 9
nodo 1
nodo 3
nodo 5
percorso 10
nodo 1
nodo 3
nodo 5
nodo 6
percorso 11
nodo 2
nodo 4
percorso 12
nodo 2
nodo 4
nodo 6
percorso 13
nodo 2
nodo 5
percorso 14
nodo 2
nodo 5
nodo 6
percorso 15
nodo 2
nodo 1
percorso 16
nodo 3
nodo 4
percorso 17
nodo 3
nodo 4
nodo 6
percorso 18
nodo 3
nodo 5
percorso 19
nodo 3
nodo 5
nodo 6
percorso 20
nodo 3
nodo 1
percorso 21
nodo 4
nodo 6
percorso 22
nodo 4
nodo 2
nodo 1
percorso 23
nodo 4
nodo 3
nodo 1
percorso 24
nodo 4
nodo 2
percorso 25
nodo 4
nodo 3
percorso 26
nodo 5
nodo 6
percorso 27
nodo 5
nodo 2
nodo 1
percorso 28
nodo 5
nodo 3
nodo 1
percorso 29
nodo 5
nodo 2
percorso 30
nodo 5
nodo 3
percorso 31
nodo 6
nodo 4
nodo 2
nodo 1
percorso 32
nodo 6
nodo 5
nodo 2
nodo 1
percorso 33
nodo 6
nodo 4
nodo 3
nodo 1
percorso 34
nodo 6
nodo 5
nodo 3
nodo 1
percorso 35
nodo 6
nodo 4
nodo 2
percorso 36
nodo 6
nodo 5
nodo 2
percorso 37
nodo 6
nodo 4
nodo 3
percorso 38
nodo 6
nodo 5
nodo 3
percorso 39
nodo 6
nodo 4
percorso 40
nodo 6
nodo 5

Figure 2: Complete list of paths in the example graph

In Appendix E we report the simplified code of the Paths Manager setup and of methods that manage the creation of paths: those methods are described here in pseudo code.

```
//-----------------------------------------------------------------------------
//                       definition of function main()
//-----------------------------------------------------------------------------

  // ordered list with all the paths
  private List paths_list;

  // working list, with the reversed paths (to be added to the final list)
  private List paths_list2;

  // list of Stop "Nodes"
  private List stop_nodes;

protected void main(){
    // read from database the data and organize them in the over mentioned lists
    readPaths();

    // the agent suspends and waits for a new message from a train.
    // when a message is received the method receivedMessage(message m) is invoked
    sleep();
 }


// -----------------------------------------------------------------------------
//             definition of procedure receivedMessage(ACLmessage msg)
//-----------------------------------------------------------------------------

void receivedMessage(ACLmessage msg){
    // this procedure is automatically called when a sleeping agent receives a new message

    // the next procedures parse the content of msg and return the requested information.
    // the format of the content is described in the previous sections
    String priority, first_node, last_node, stop_node, used_path, skip_node;

    priority = msg.getPriotiry(msg);
    first_node = msg.getFirstNode(msg);
    last_node = msg.getLastNode(msg);
    stop_node = getStopNode(msg);
    used_path = getCurrentPath(msg);
    skip_node = getSkipNode(msg);

    // call the method that finds the alternative paths and sends back them to the train
    getAlternativePaths(msg, first_node, last_node, stop_node, priority, used_path, skip_node);
 }


// -----------------------------------------------------------------------------
//             definition of procedure readPaths()
//-----------------------------------------------------------------------------

private void readPaths(){

    ResultSet srs = executeQuery("SELECT id FROM nodi where sosta= true order by id");

    nodi_sosta_= new ArrayList();
    while (srs.next()) stop_nodes.add(String(srs.getInt("id")));
```

```
    srs = executeQuery("SELECT id FROM nodi order by id");

    paths_list= new ArrayList();
    paths_list2= new ArrayList();

    while (srs.next()) do

        List path= new ArrayList();
        path.add(String(srs.getInt(1)));
        dfs(srs.getInt("id"), path);

    end do

    // this method moves every element of the list paths_list2
    // into the final list paths_list,
    // ordering all paths with respect to the name of the first node of the path
    merge_percorsi();

  }

// ------------------------------------------------------------------------------
//             definition of procedure dfs(int id_nodo, List percorso)
//------------------------------------------------------------------------------

  private void dfs(int id_nodo, List path){

    ResultSet srs = executeQuery(
    "SELECT distinct id_nodo_uscente, id_nodo_entrante, entrata, uscita " +
    " FROM itinerari inner join nodi on nodi.id=itinerari.id_nodo_entrante " +
    " WHERE id_nodo_uscente= " + id_nodo + " order by id_nodo_entrante");

    if(path.size()>1) then
        paths_list.add(path);

        // note that this code is correct if, and only if,
        // we assume, as in "Stand-alone FYPA", that all arcs are bidirectional
        List reversed_path= new ArrayList();
        for(int i= percorso.size(); i>0; i--) do
            reversed_path.add(path.get(i-1));
        end do

        paths_list2.add(reversed_path);

    end

    while (srs.next()) do

        List npath= new ArrayList();
        for(int i= 0;i<path.size();i++) npath.add(i, (String)path.get(i));

        npath.add(String(srs.getInt(2)));

        // note that this code is correct only if we inserted in the database
        // all arcs in one traveling direction!
        dfs(srs.getInt(2), npath);

    end do
}
```

```
// ------------------------------------------------------------------------------
//  definition of procedure getAlternativePaths(ACLmessage msg, String first_node,
      String last_node,String stop_node,String current_path, String skip_node)
//------------------------------------------------------------------------------

private void getAlternativePaths(ACLmessage msg,
String first_node,String last_node,String stop_node,String current_path, String skip_node){

    // parse the parameter lista that contains the current path of train
    // (to be excluded from the returned list)

    // this function, described in Appendix E, returns all the paths connecting
    // the first_node and last_node but not containing skip_node, avoiding to return the current_path too
    List ret=find_paths(first_node, last_node, stop_node, current_path, skip_node);

    String txt="";
    for(i = 0;i < ret.size();i++){
        // this function add to the first parameter a new String representing
        // an alternative path, in the format described in the previous sections
        txt = add_alternative_path(txt, ret[i]);
    }

    ACLMessage rep = msg.createReply(ACLMessage.INFORM);
    rep.setContent(txt);
    send(rep);
  }
```

5.5 NetLogo: creating a post-execution simulation for "Stand-alone FYPA"

When the "Ansaldo FYPA" system was designed and implemented, we tested and executed it using the data and the software from Ansaldo STS. The external system of Ansaldo STS is made by several modules, and an interface that shows the station and the trains movements was already available. The engineers of Ansaldo STS were able to connect their system with "Ansaldo FYPA" system and to show, in real time, how the simulation was going on. Two screen shots taken from the Ansaldo STS interface are reported in Figure 3 and 4: the images show the trains position at the end of simulation (considering the Nodes on the y-axis and Time on the x-axis), and their colors represent what type of change the FYPA system imposed on trains (none, only time changes, path change, arc change and so on). This interface is updated at run-time, while the system is executing: at the top of the images, time (expressed as a number between 0 and 24, divided in 30 intervals of 2 minutes) moves on and trains are moved on the graph and colored to show the user how the system is acting.

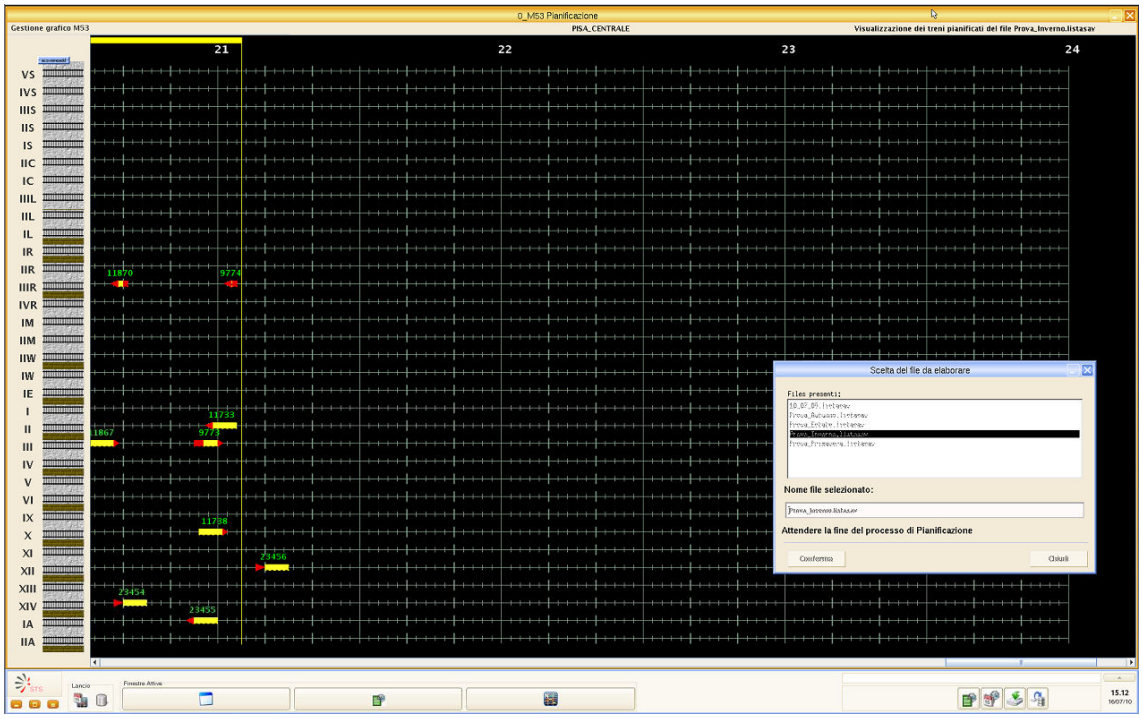


Figure 3: Screen shot 1 from Ansaldo STS interface

When we started working on the “Stand-alone FYPA” system we did not have this graphical interface anymore. Anyway, we though that a graphical representation of the simulation could be useful to us too, because JADE only offers the sniffer to see how the system is acting, but this remains a representation of the messages exchanged, not a concrete representation of the agents or of the station. Without a graphical interface the only way we had to test our system was to use the output logfiles (that remains the most powerful way to control the overall execution but that can be really complex to analyze) and the sniffer. To make our system in some way more “user friendly” we decided to add to the system a NetLogo program (not integrated in JADE, an external software application) that was able to take as input the logfiles from FYPA, after its execution, and was able to simulate in a graphical way the evolution of the simulation, that is, to represent in a schematic way the station and the movements of the involved agents.

We choose NetLogo because, as described in Section 2.3.3, it lets the programmer simply create a graphical representation of agents. The solution of integrating in our system a graphical interface (made using Java) was not possible to follow because it would required too time to be developed: being this interface only a plus for our system, we preferred using that time to improve our protocol and to deepen our research.

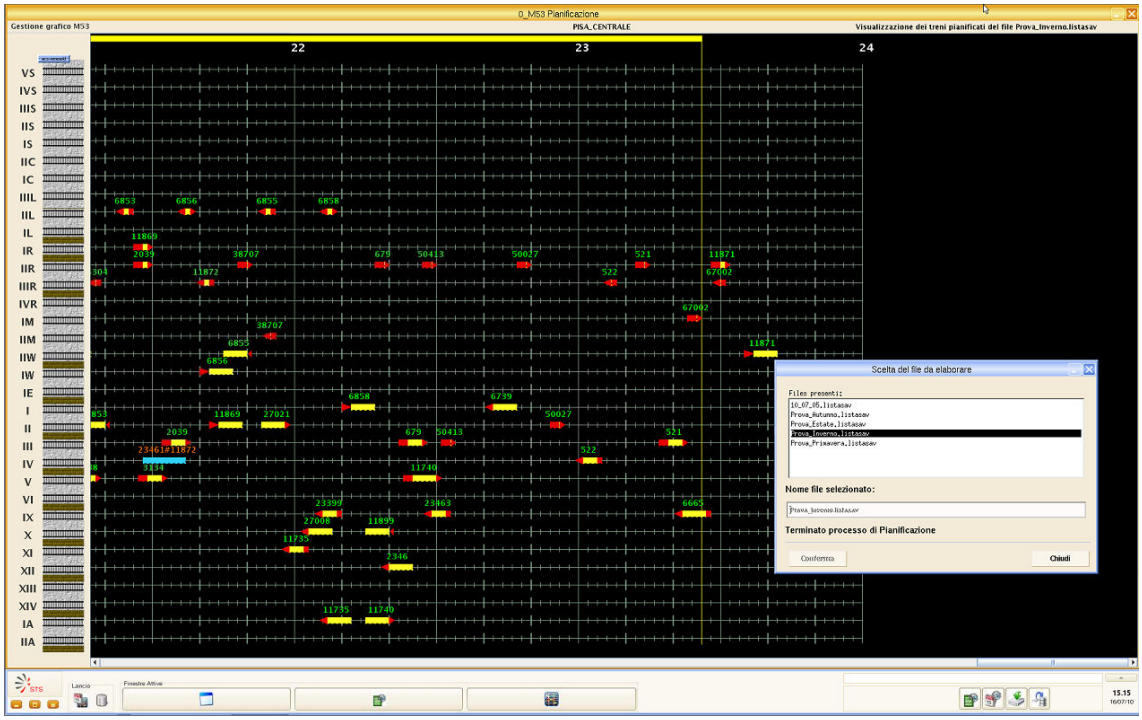


Figure 4: Screen shot 2 from Ansaldo STS interface

5.5.1 NetLogo as programming language for FYPA

The NetLogo environment (and programming language) is really interesting and we also evaluated to map our algorithm using it, but after a deep analysis we understood that this translation was neither simple nor really useful, as described later on.

We decided not to use NetLogo to translate our algorithm because its main features are not so suitable to simulate a negotiation among agents based on a "deterministic view" (that is in this case, a negotiation based on a protocol with predefined roles and actions), nor to simulate an interaction where agents have to exchange messages.

NetLogo is in fact very useful to simulate the evolution of a system made up of several agents, usually divided into categories, where there is no need to follow the life of a specific agent, but instead the main aim is to evaluate the emergent behavior of the system. Moreover, in this kind of simulation usually decisions about what action is to be done are made using a probabilistic choice. Moreover the concept of time is not simple to insert, if the programmer needs something more complex than the built in representation of time with $Ticks$ (NetLogo integrates a discrete representation of time, that is shared among agents).

In our algorithm we needed to represent a graph, with nodes and many arcs connecting them,

and agents that cross the graph. In NetLogo $Links$ (built in type of agent), that clearly represent a connection between two agents, exist, but only one $link$ for every pair of agents can be created. So the mapping of the multiple connections among two nodes was difficult to achieve. Furthermore, the most natural way of using NetLogo is to ask the entire set of a kind of agent to do something, using the command "ask $agentset$ to do something": unfortunately this command selects from the set an agent in a random way and than execute the code for this agent. Then it chooses randomly another agent and so on, till all the agents have been selected. In this way, the execution of the entire protocol is sequential, and not simultaneous, so all the problems connected to the simultaneous choice of a path, arc or node to move on does no more make sense.

We must underline that there are ways to avoid this behavior of the "ask command", but considering the main features of this language it is clear that to simulate exactly what we wanted was very hard and time consuming, because this is not the right language to simulate what we were looking for. Furthermore, in NetLogo agents are not allowed to exchange messages directly, and above all is quite "out of the NetLogo philosophy" trying to discriminate among agents of the same type: we need to have many Treno agents with very different plans and goals, and every agent has to act as an individual, not as all the others, and this is hard to map in NetLogo.

5.5.2 The developed NetLogo graphical interface

NetLogo environment offers an integrated graphical representation of the simulation, so it is quite simple to let the user see how the simulation is going on. Instead of programming the graphical front end of an application, to see for example where the agents are moving, using Java Api for drawing, in NetLogo you have this for free.

Our NetLogo software reads from input files the structure of the station and the movements of a set of Trains during the time. Due to NetLogo limitations, we are not able to draw more that one arc connecting two nodes.

Using the NetLogo terminology, nodes and trains are represented as a particular type $Turtles$, while arcs are NetLogo $Link$ agents. The time is represented using NetLogo $Ticks$: at every tick the trains will read from a private list (filled with their movements on the graph) and will move on a new node (or will remain on the node where they already are).

Due to time restriction we did not have enough time to add to the "Stand-alone FYPA" the Java code to produce automatically the output files representing the simulation in the format we foreseen for NetLogo. At the time when this thesis was written the input files for the NetLogo graphical interface are manually filled (as an extract from the "Stand-alone FYPA" log files).

The graphical interface needs four types of files:

- Nodi.txt: this is a list of the nodes (with their physical position on the NetLogo output screen);
- Archi.txt: this is a list of the arcs connecting two nodes (at max one for each pair);
- Treni.txt: this is the list of all the trains involved in the simulation;
- Movimenti.txt: this is a list representing, for each $tick$, where the trains are.

In Figure 5 the interface of our NetLogo program is reported.

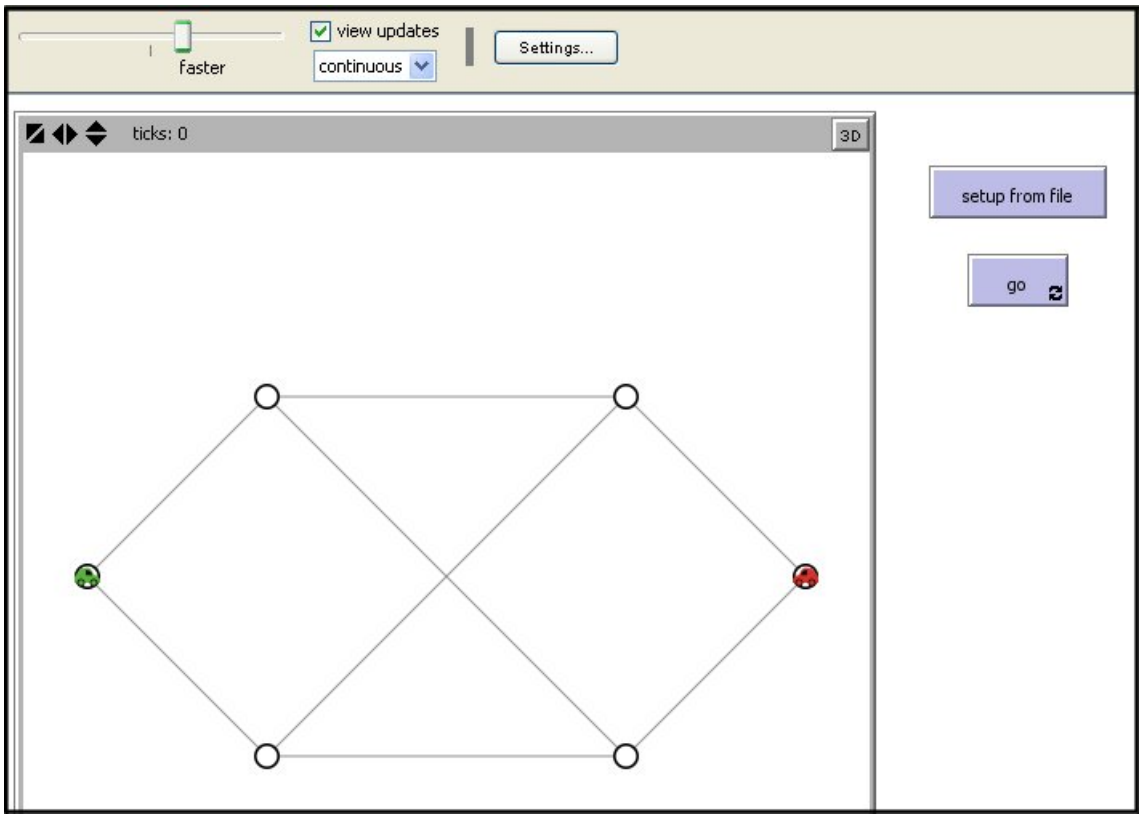


Figure 5: The graphical interface made with NetLogo

When the graphical interface is started, the agents will read from the above described files the information they need. The user can decide if running the simulation "tick by tick" or "as a movie" (selecting the flag "Forever" among the options of the "Go" button): in the first case the user will ask the system, pressing the button "Go", to move on the simulation of only one tick (updating the trains position), while in the second case the system will update the graph every

$Delta$ seconds (that is, every $Delta$ seconds the tick counter will be incremented and the trains position updated), showing in this way the trains moving on the graph. $Delta$ is a value that the user can change to slow down or accelerate the simulation, using the sliding bar shown in the top of the interface.

5.6 Evaluation of the "Stand-alone FYPA" system

The "Stand-alone FYPA" system has been tested considering different configurations, graphs and conflicts among trains: some of these tests are reported later in Section 6. All the basic conflicts (those involving one or two trains or regarding the management of one node) are solved by the MAS within two seconds. The more complex configurations, representing real stations, have been tested by Ansaldo STS using the "Ansaldo FYPA" system. The "Ansaldo FYPA" system shows good performances when tested by Ansaldo STS engineers: using a station with about fifty nodes, two hundreds arcs and at least ten trains in the station at the same time, the system is able to find a solution in few seconds.

Ansaldo STS is using this system as a plug-in of its current system, and "Ansaldo FYPA" is exploited to organize the trains movements in a station for an entire day. More in detail, Ansaldo STS uses a system able to calculate, twice a year, the global distribution of trains in the various Italian stations but this system does not calculate the trains allocation inside the stations. This allocation was made by the human operators. "Ansaldo FYPA" system first of all fills this gap: the output of the already existing Ansaldo STS system became the input of the FYPA system (in the "off-line version" of our MAS, so called by Ansaldo STS). The output of "Ansaldo FYPA" off-line version is the complete allocation of trains to be used in the station or, in other words, this output is the set of the original plans of trains for a specific day.

Then, "Ansaldo FYPA" system can be used on-line to reallocate trains after delays or node failures, that is, it will receive, thanks to web services that it uses to connect to the Ansaldo STS external systems, information and real data that cause conflicts in the station. These conflicts will be solved by FYPA and its solution will be presented to the end-user of the Ansaldo STS system.

Ansaldo STS kindly gave us some information about the stations actually managed using "Ansaldo FYPA" system: the execution times, that we report later on, refers to the execution of the off-line version of "Ansaldo FYPA".

The first station considered during the development of the "Ansaldo FYPA" system (and today managed with this system) is "Mestre": this station has 59 Nodo agents, 528 Treno agents (during day 28, March 2011) and each Nodo agent manages approximately tree entering arcs. The total number of incompatibility is 430. Due to the very high number of nodes and arcs, we are not able to report the graph of the station, but we report the simplified representation of the "Mestre" station (Figure 6), that is the one that Ansaldo STS uses to represent a station.

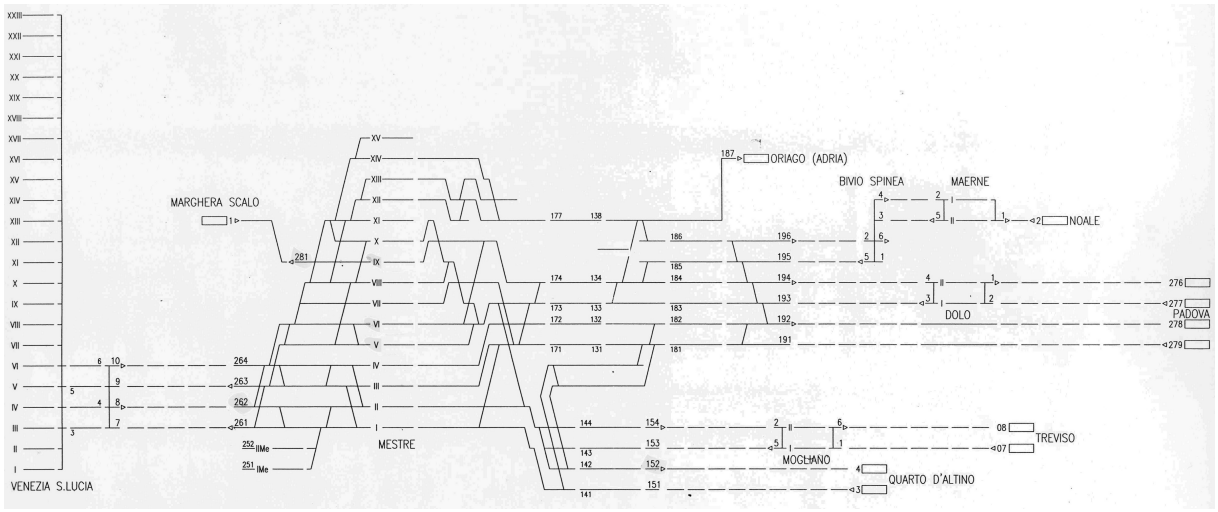


Figure 6: The simplified representation of “Mestre” station

The simulation of the chosen day takes 18 minutes to be completed. This time is also due to the scheduling chosen by the Ansaldo STS system: the User Agents Manager creates a new train every 2 seconds, so the simulation time is at least equal to the number of trains multiplied for 2. In this example, at least 17 minutes. This means that the system requires only one minute, considering all the simulation time, to manage the reallocations/conflicts that arise between trains.

The second station that is using “Ansaldo FYPA” is “Pisa”. “Pisa” is managed by 60 Nodo agents, each managing approximately 20 entering arcs. Every arc has on the average 130 incompatibilities. This station is more complex than the previous because there are more arcs and more incompatibilities. To simulate day 20 of January 2011, with 395 trains that cross the station, the simulation takes 13 minutes to be completed. A graph-like representation, provided by Ansaldo STS, is reported in Figures 7 and 8, where the station is divided in left and right sub-graph.

The “Stand-alone FYPA” performs in the same way if considering the “allocation phase”, that is, if considering the simulation starting when the first train enters the station. “Stand-alone FYPA” is faster in its start up phase (creating the Nodo agents with the information regarding arcs and incompatibilities) than “Ansaldo FYPA” because the direct access to a database is really faster than the usage of web services and the parse of very large files. The “Stand-alone FYPA” system takes about thirty seconds to set up, while the “Ansaldo FYPA” needs some minutes.

Some improvements we could make to the “Stand-alone FYPA” software are:

- Completing the code to automatically export the log files in the format requested by our NetLogo graphical interface.
- Modifying the database (and the code) to model directed arcs instead of bidirectional arcs.

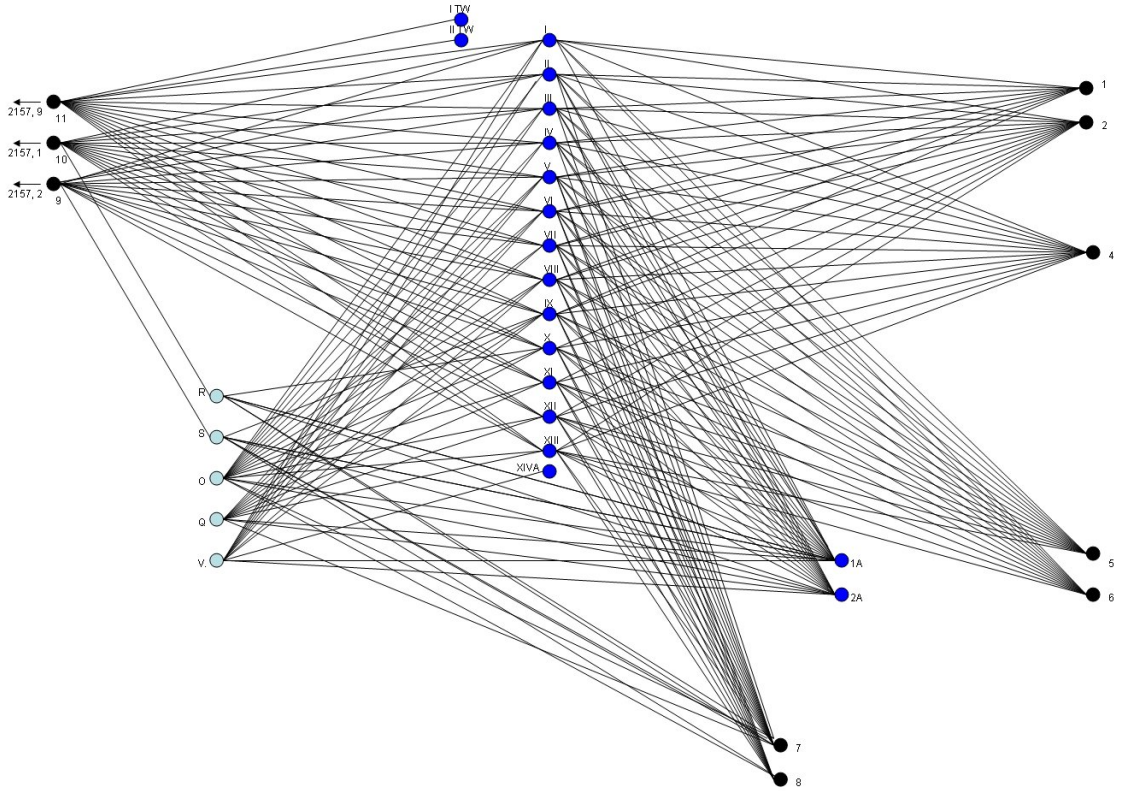


Figure 7: The graph-like representation of “Pisa” station (right side)

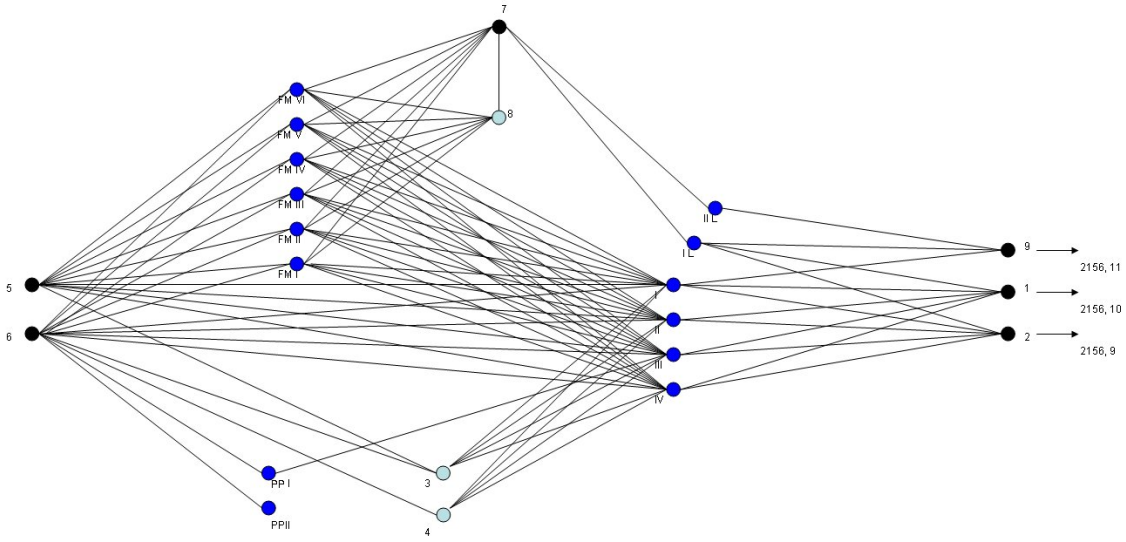


Figure 8: The graph-like representation of “Pisa” station (left side)

- Modifying the protocol to let Treno agents "ask help": if a Treno agent $T1$ needs a resource that is already reserved by another Treno agent $T2$ with an higher priority (so $T1$ is not able to steal the resource), $T1$ could ask $T2$ to find another path in the station, that is, it could ask $T2$ to release the resource and let $T1$ move on that.
- Inserting the configuration parameters regarding the "time outs" into the database, in order to allow the user specify how long a Nodo or a Treno agent must wait for an answer.
- Adding a "Creator Agent" that decides the "Zero time" of the simulation and then creates the manager agents.

Furthermore, we would like to look for similar case studies in the same area to verify if FYPA may be used to solve them: by analyzing these new case studies we will also be able to understand how our protocol can be modified to cope with them and, in general, how FYPA can be improved.

We will also consider the possibility of integrating ontologies in FYPA: some motivations for this integration are discussed in Section 8.7, when we will describe our proposal to integrate ontologies in MASs.

Chapter 6

FYPA: Examples

In this chapter we will describe the main features of the FYPA protocol using some examples: we will start with simple ones to show the basic interactions in the negotiation protocol, how the changes of the state of the graph are spread among Nodes, the node reservation procedure and so on. Then we will present some more complex examples (using sometimes a Dummy agent from JADE platform) to show the interactions among Trains and Nodes.

For each example we will report the original plans of Trains and the graph configuration: then we will add the main outputs of the system or the most interesting exchanged messages, and the JADE'sniffer view, if the figure is enough readable. Note that in plans of Trains time is expressed in milliseconds: we did not change the data type in the thesis to let the reader find a mapping between the data in the original plans and those in the messages exchanged by the agents. Furthermore, every line in a plan is a step of the Train on the graph: we use the column "Step" to let the reader simply identify the order in the path, without comparing the "From" values.

Moreover, to fully understand the examples it is important to remember that the FYPA system uses different names for the agents, in particular:

- every User Agent (UA) is called "Treno"
- every Resource Agent (RA) is called "Nodo"
- the Resource Agents Manager is called "PrenotazioniStazione"
- the User Agents Manager is called "MovimentiIndotti"
- the Paths Manager Agent (PA) is called "PercorsiAlternativi"

In the examples we will use the FYPA system names, to make simpler to understand Figures and the content of ACL messages.

These examples have been executed on a personal computer with MS Windows XP Professional©, 2 GB RAM and an AMD Athlon© 64 processor 3000+.

6.1 Resource reservation

In this example we show how Nodo agents interact to maintain the state of the graph updated.

Considering the graph as shown in Figure 1, $Nodo\_3$ and $Nodo\_4$ manage the arc number 6 which has an incompatibility with arc 5, managed by $Nodo\_2$ and $Nodo\_5$. These arcs are managed by two Nodo agents because they are bidirectional.

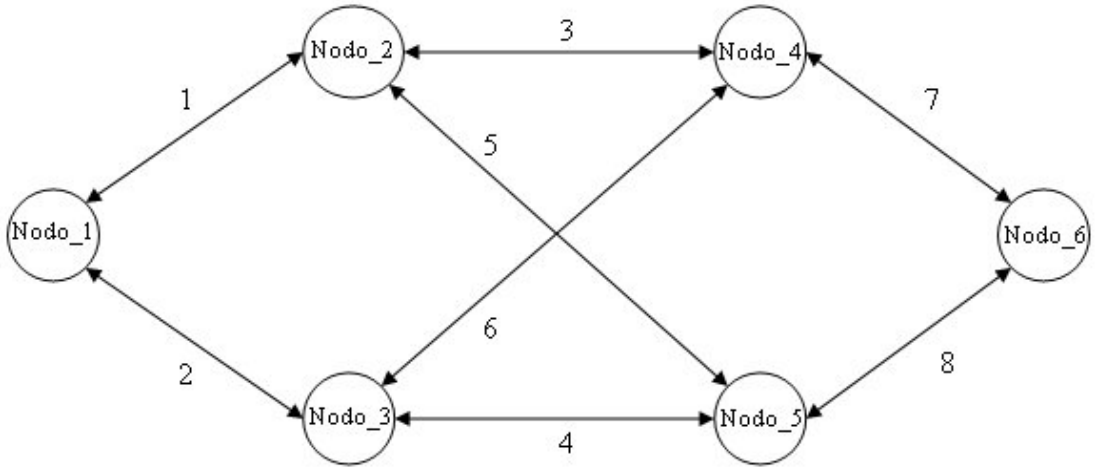


Figure 1: Graph architecture

Let us simplify the example using a Dummy agent ($da0$) instead of a Train and let us suppose that this agent needs to leave $Nodo\_3$ to move on $Nodo\_4$. In this case $da0$ will send a reservation request to $Nodo\_4$ and then a confirmation message to it. In Figure 2 the QUERY-IF message sent by $da0$ to $Nodo\_4$ is reported with the answer sent by $Nodo\_4$ to $da0$, confirming that the resource is free and that $da0$ can use arc 6.

In Figure 3 the messages exchanged between Nodo agents are reported: $Nodo\_4$ informs $Nodo\_2$, $Nodo\_3$ and $Nodo\_5$ that there is a reservation request they must be informed of, and then it sends a new message, with performative CONFIRM, to inform its neighbors that the previous reservation request has been confirmed.

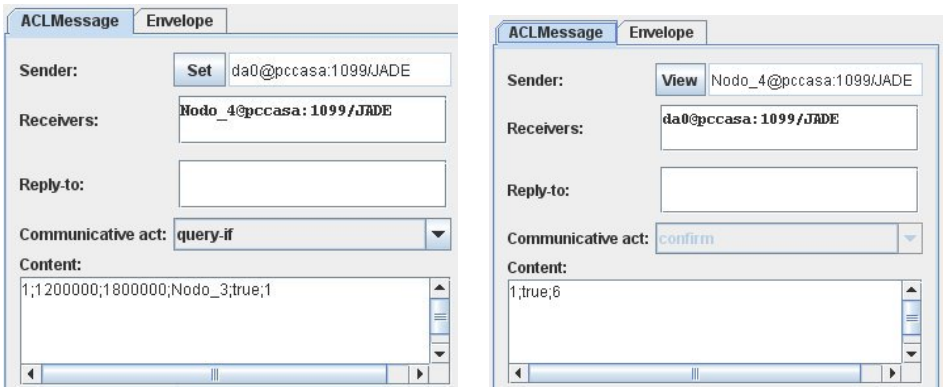


Figure 2: QUERY-IF and CONFIRM messages exchanged between $Nodo\_4$ and $da0$

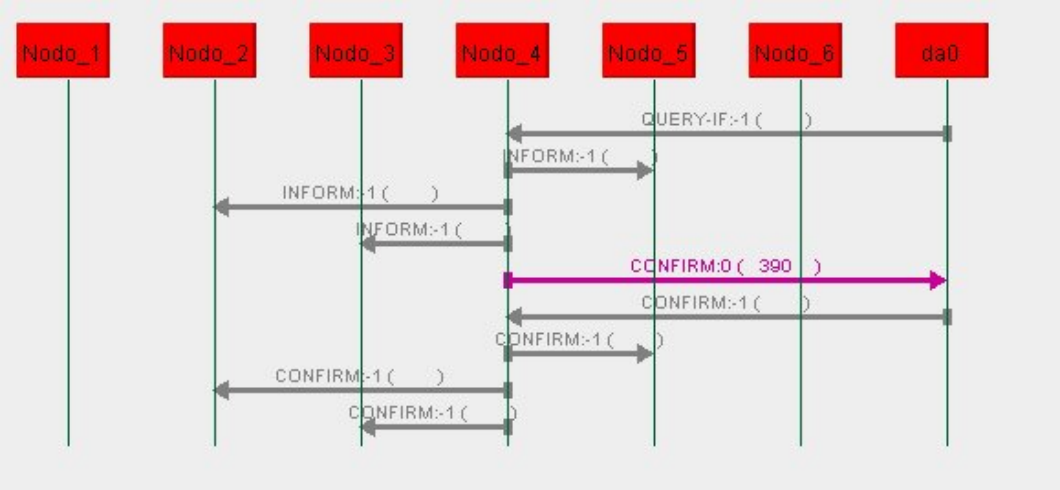


Figure 3: Sniffer view

In Figure 4 the INFORM messages sent by $Nodo\_4$ are reported. Their content is different (only the first parameter) for each Nodo, reflecting the different arc they refer to:

- For $Nodo\_2$, the list reports that $Nodo\_2$ must update the state of arc 5, exiting from $Nodo\_5$ (this because arc 5 is the one with an incompatibility with arc 6)
- For $Nodo\_5$, the list reports that $Nodo\_5$ must update the state of arc 5, exiting from $Nodo\_2$ (as for the previous point, but in this case the arc, that is bidirectional, exits from $Nodo\_2$)
- For $Nodo\_3$, the list reports that $Nodo\_3$ must update the state of arc 6, exiting from $Nodo\_4$ (being arc 6 bidirectional, its status must be updated for both the nodes it connects)

The execution of this example in JADE takes less than 1 second to be completed.

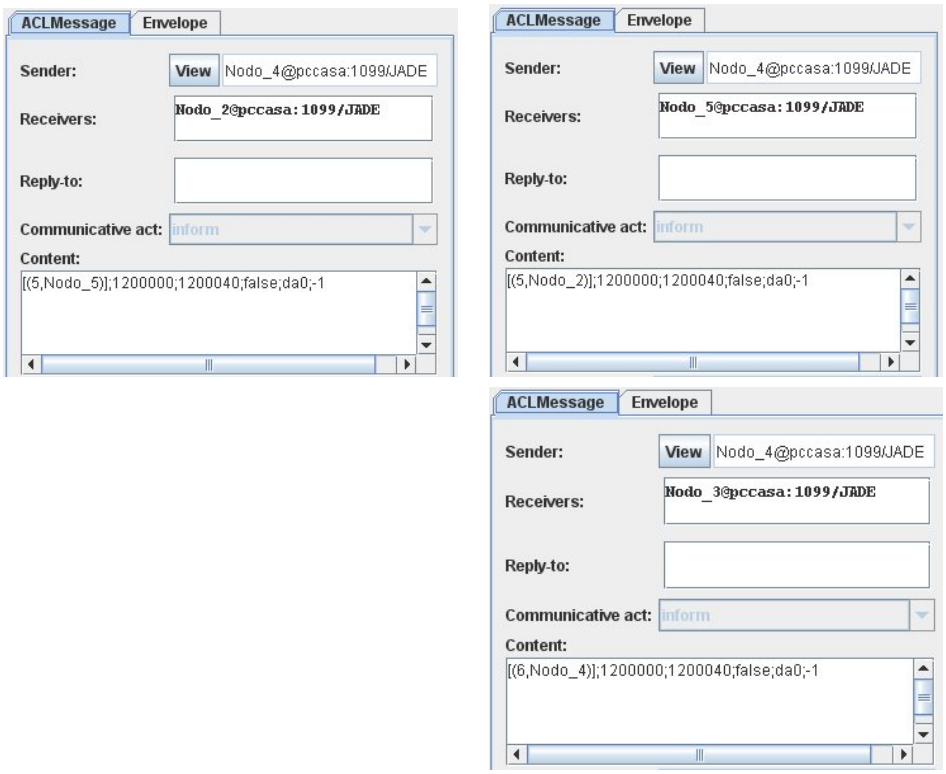


Figure 4: INFORM messages sent from $Nodo\_4$ to its neighbors

6.2 Path reservation: free path

In this example we describe the procedure to reserve a path, and we start from the simplest situation, that is, all the nodes are free and there is only one train in the station. In the next sections we will present more complex examples.

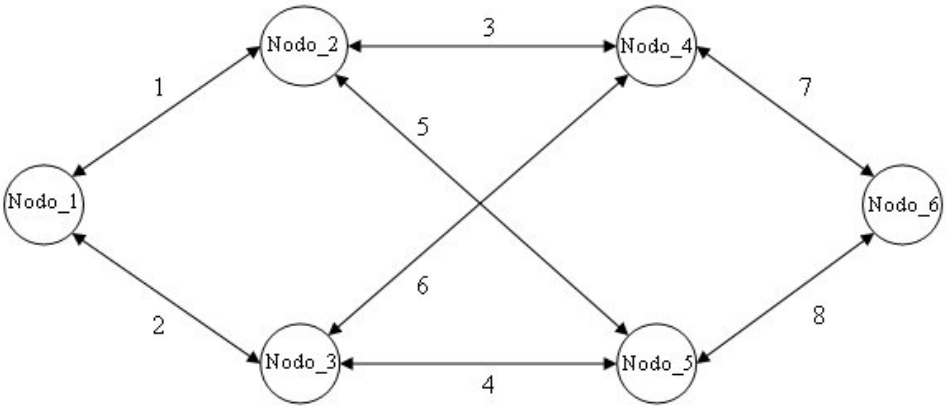


Figure 5: The station

In Table 1 the original plan of $Treno\_1$ is reported: considering the station in Figure 5, $Treno\_1$ needs to reserve the path *Nodo_1, Nodo_3, Nodo_4 and Nodo_6*. In Figure 7 the complete interaction between $Treno\_1$ and nodes is shown.

Train	**Step**	**Node**	**From (ms)**	**To (ms)**
Treno_1	1	Nodo_1	210000	240000
Treno_1	2	Nodo_3	240000	310000
Treno_1	3	Nodo_4	310000	340000
Treno_1	4	Nodo_6	340000	380000

Table 1: $Treno\_1$ original plan

In Figure 6 we report the four QUERY-IF messages sent by $Treno\_1$ to the Nodo agents. This interaction is quite simple: all the Nodo agents have no other request reservations so they all confirm the reservation request of $Treno\_1$, that will again confirm them when it receives all the answers by the Nodo agents.

In Figure 8 the movements of $Treno\_1$ in the station are reported: these images come from our plug-in made with NetLogo, as describe in 5.5. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 1 second to be completed.

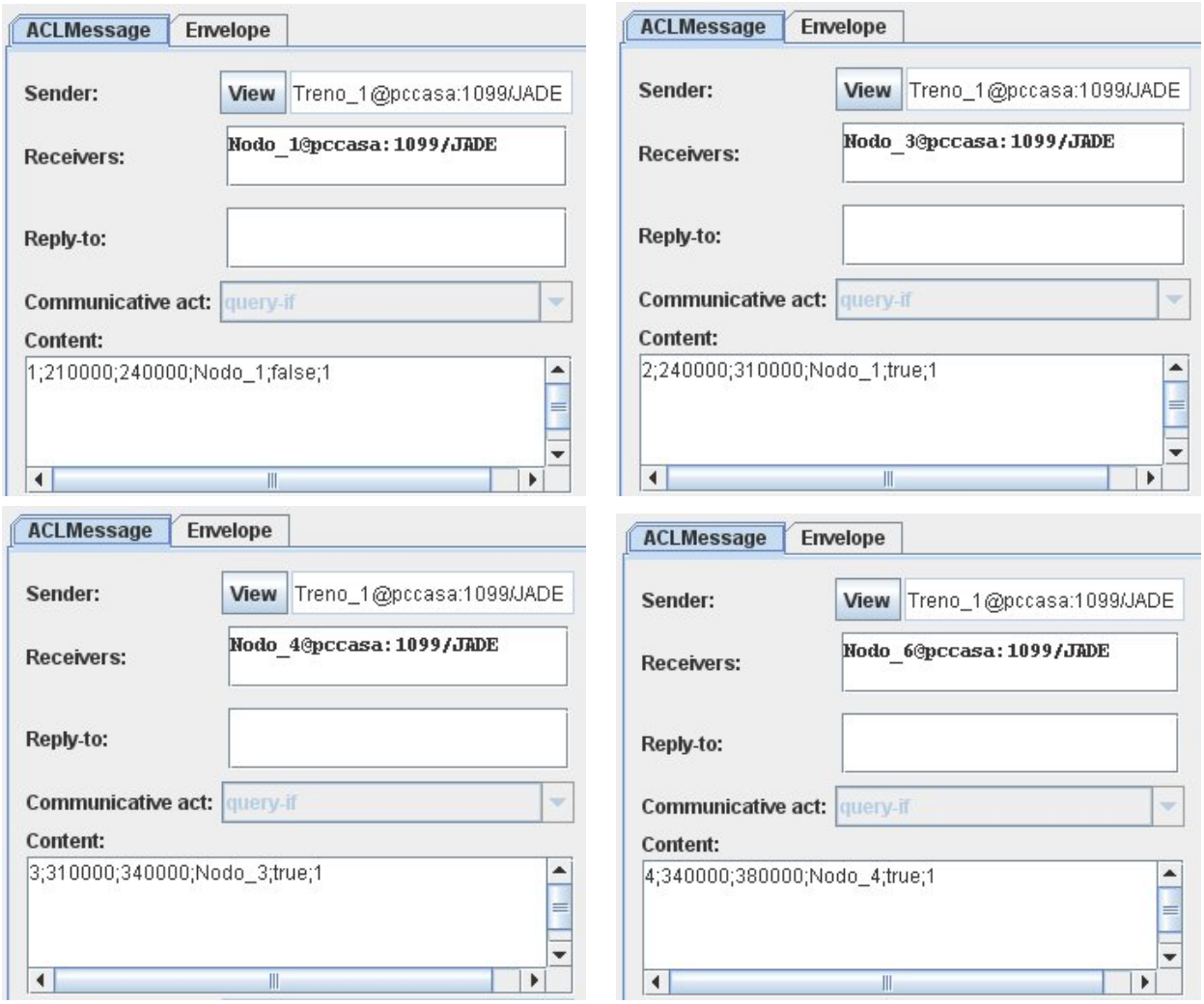


Figure 6: QUERY-IF messages sent from $Treno\_1$ to the Nodo agents

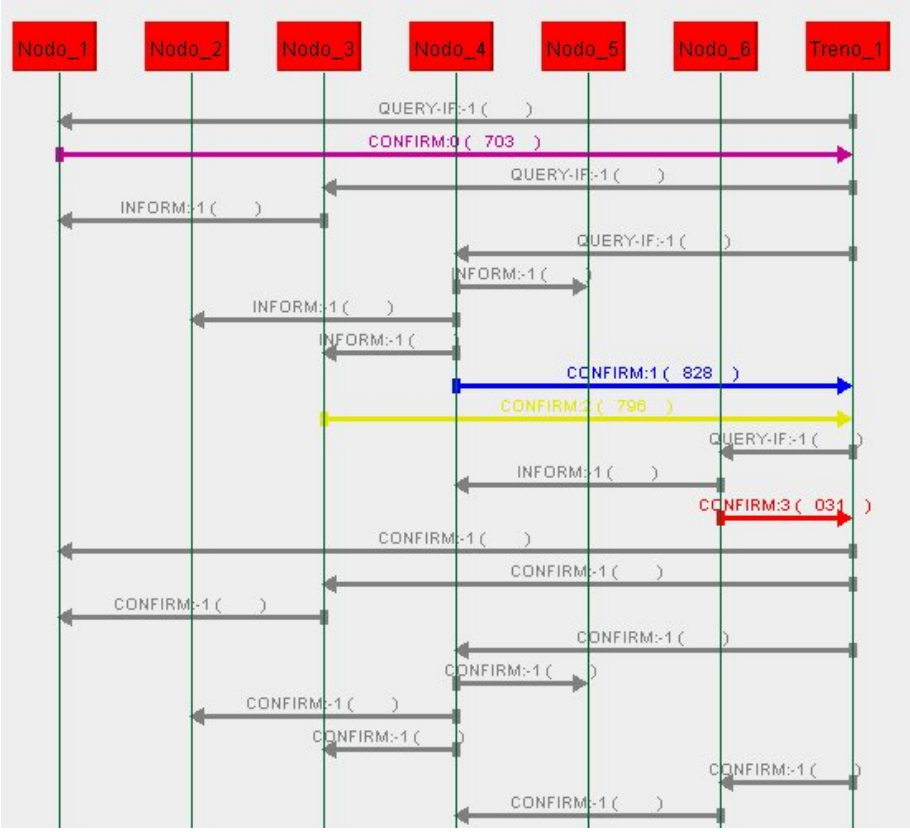


Figure 7: Sniffer view

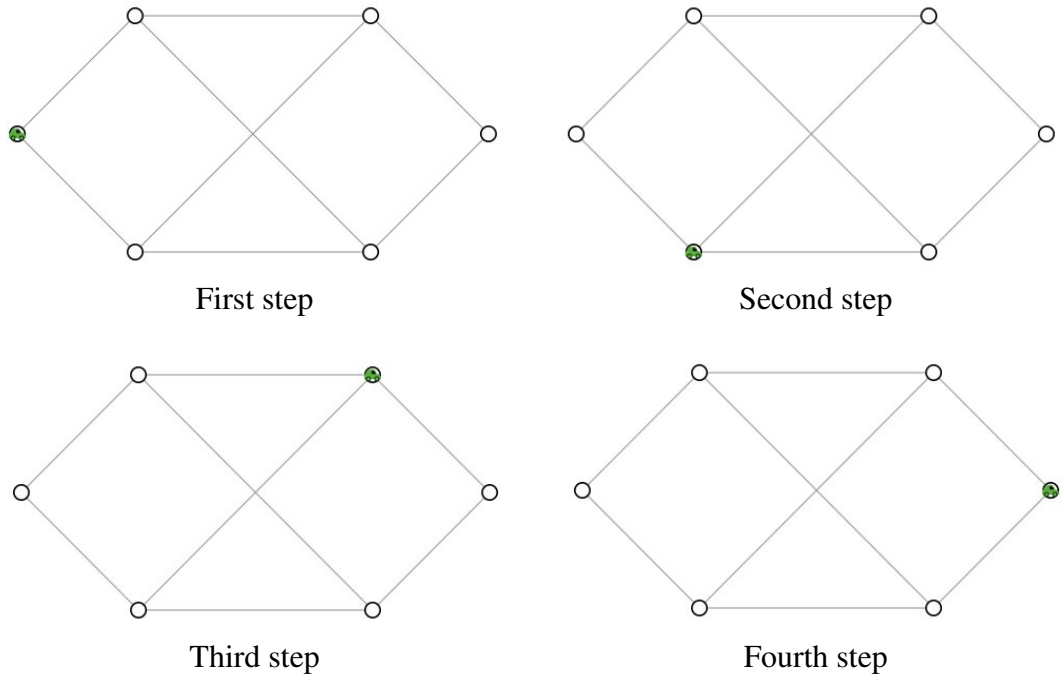


Figure 8: Screen shots from NetLogo

6.3 Path reservation: occupied Node (case 1)

Now let us complicate a little the above example. Suppose that $Treno\_1$ has priority equal to 2. The station and $Treno\_1$'s original plan are the same of the above example.

We will show how the Treno agent acts if one of the resources it is trying to reserve is already occupied.

In this example we assume that another train (a dummy agent, $da0$) has already a reservation for the resource $Nodo\_4$ for the interval 310000 - 340000 (that is the same interval requested by $Treno\_1$). The behavior of $Treno\_1$ will change considering the maximum delay it can undergo and the priority of $da0$.

Let us suppose that $Treno\_1$ can now accept 4000 milliseconds as maximum delay, and that $da0$ priority is 3 ($da0$ is a train with a lower priority than $Treno\_1$).

As the reader can see in Figure 10, when $Treno\_1$ sends it QUERY-IF messages, it receives an INFORM from $Nodo\_4$ (reported in Figure 9) that specifies that the resource is already occupied by train $da0$, with priority 3, and that the resource will be again available from 340001 (till 370001, that is, is free for the same time interval but starting from 340001).

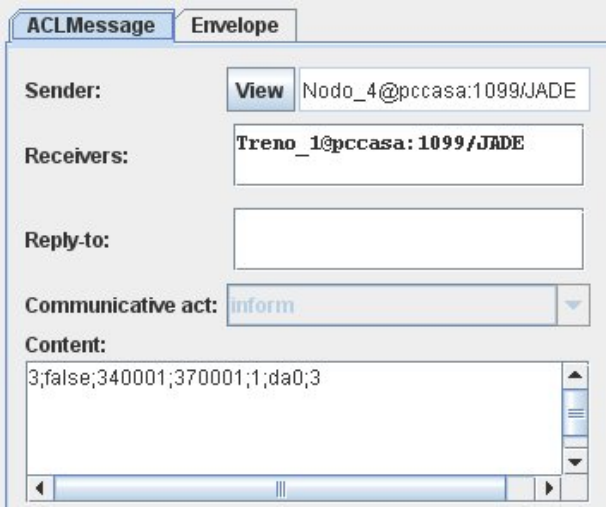


Figure 9: INFORM message from $Nodo\_4$

$Treno\_1$ can calculate the total delay it should accept: in this case, 3000 milliseconds, that is an acceptable delay. The train decides to maintain its path (even if it could steal the resource from $da0$) shifting the reservations: it asks $Nodo\_3$ to stay more on it, and then it moves on $Nodo\_4$ when specified in the INFORM message. In Figure 11 the new QUERY-IF messages sent by $Treno\_1$ are reported, to show the new time intervals chosen by the agent. All the nodes will answer with a CONFIRM, so the train will again confirm its reservations and will get a reserved path.

The interaction among the agents (excluding the last CONFIRM messages sent back by $Treno\_1$) is shown in Figure 10, while the complete simulation made with the NetLogo interface is reported in Figure 12. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 2 seconds to be completed.

In this situation, the priority of $da0$ does not care because $Treno\_1$ can accept the delay suggested by $Nodo\_4$. In the next examples we will describe a situation where $da0$'s priority will make the difference in the behavior of trains.

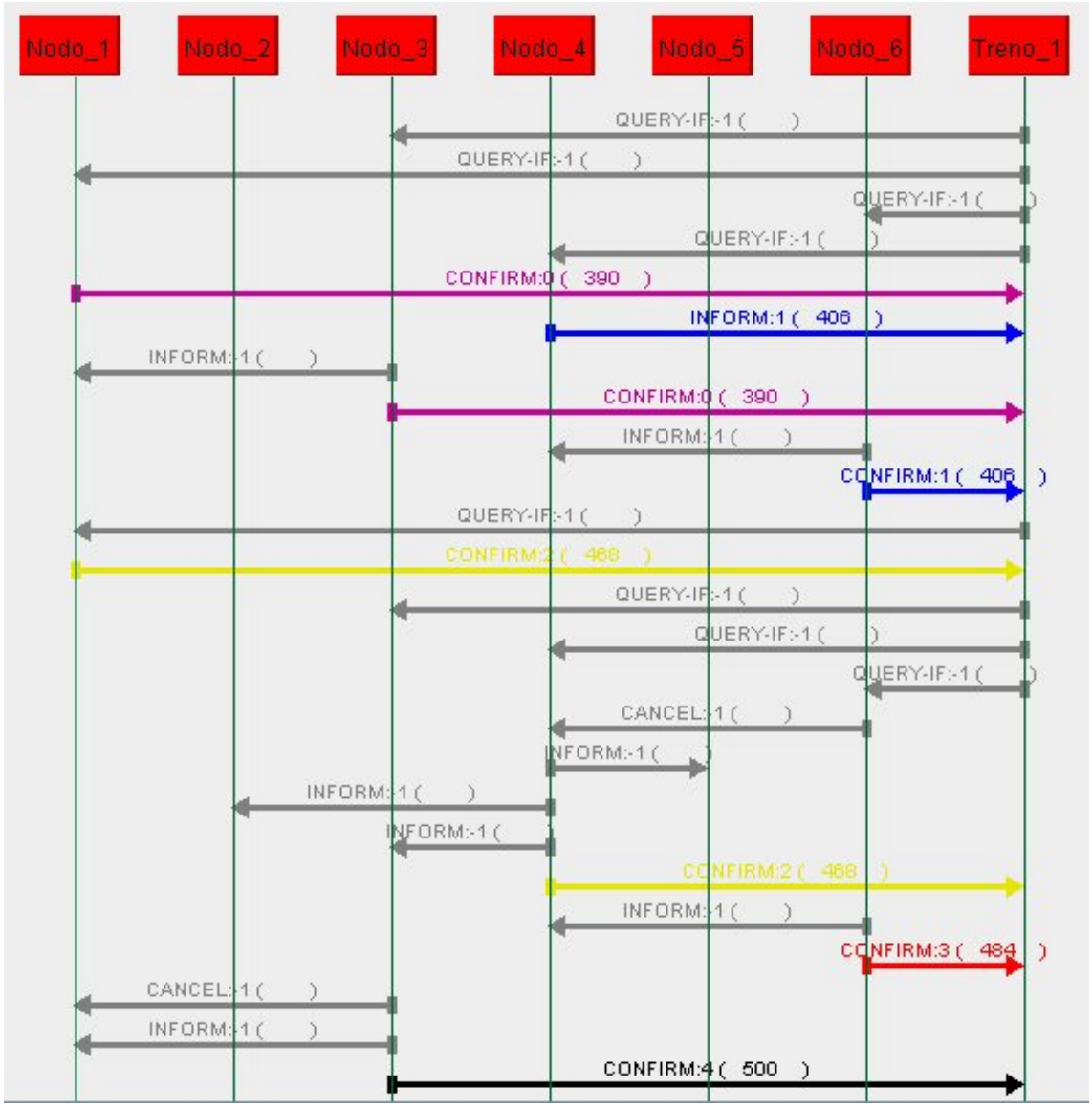


Figure 10: Sniffer view

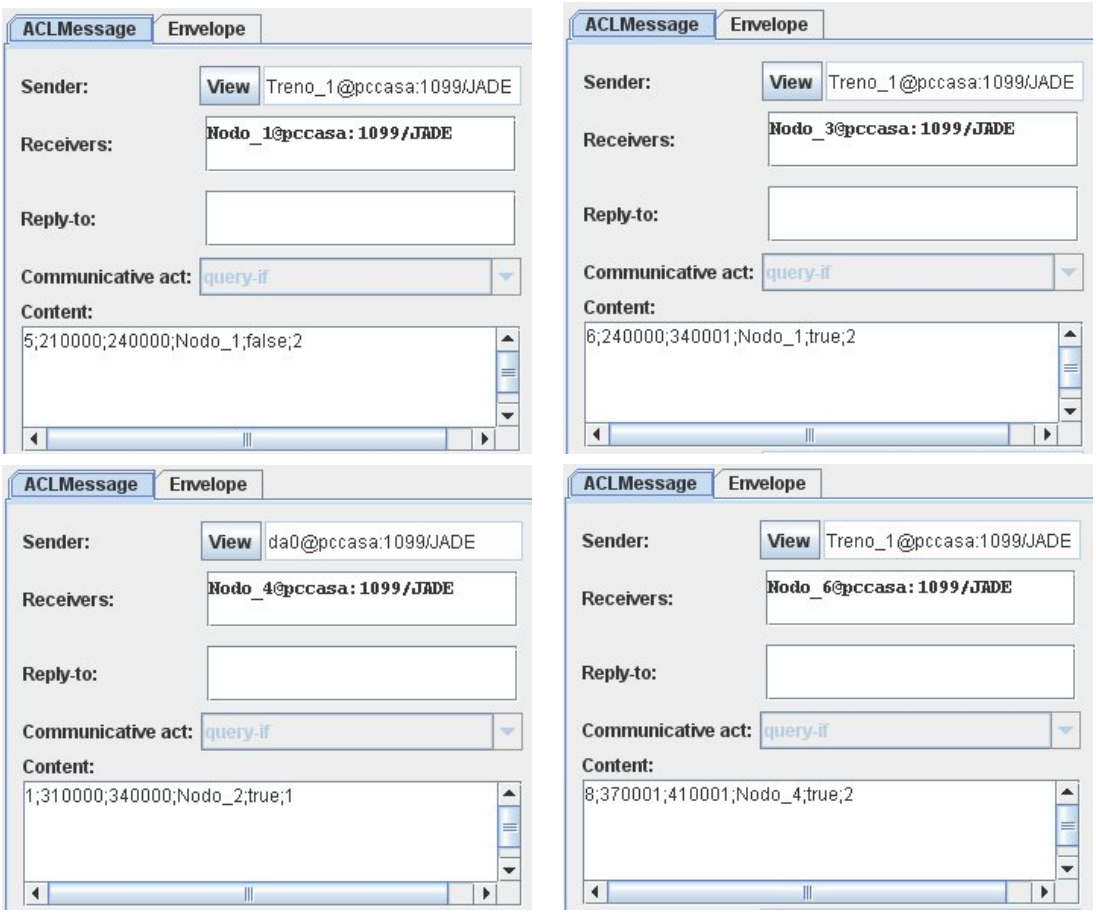


Figure 11: New QUERY-IF messages sent from $Treno\_1$ to the Nodo agents

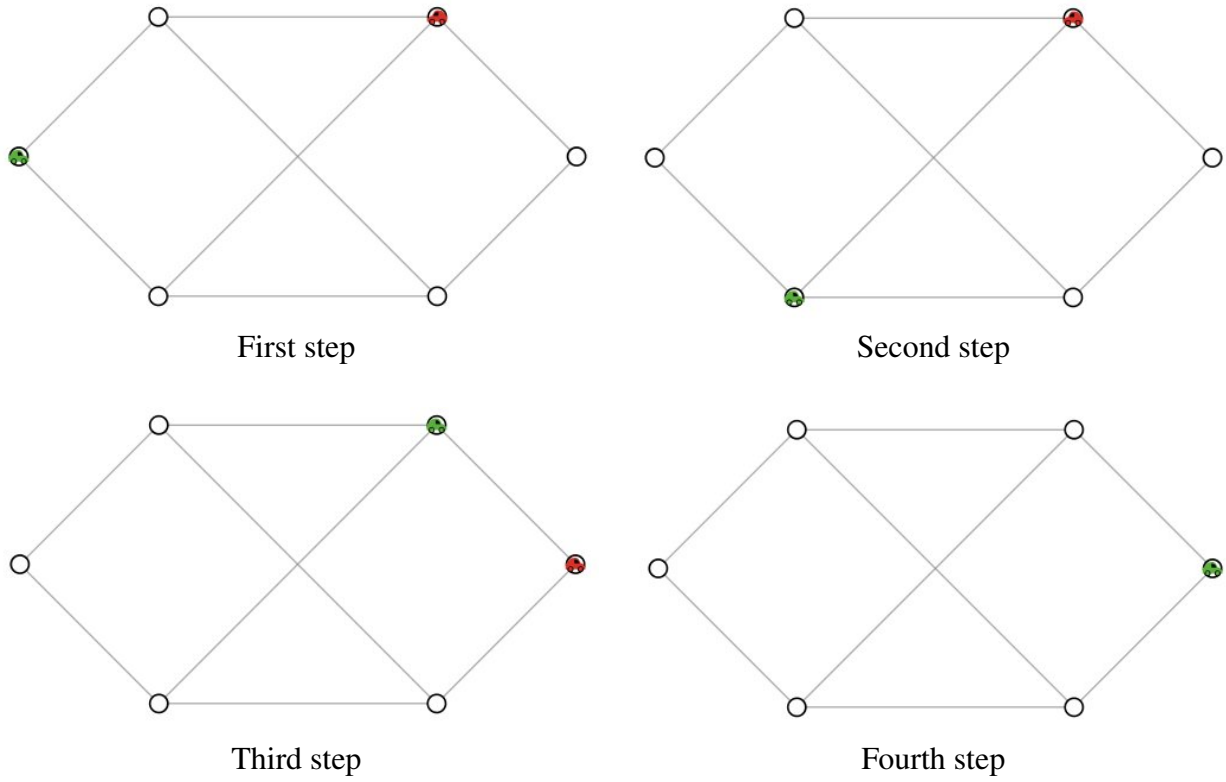


Figure 12: Screen shots from NetLogo

6.4 Path reservation: occupied Node (case 2)

In this example we assume that the dummy agent $da0$, that has already a reservation for the resource $Nodo\_3$ for the interval 240000 - 310000, has priority 3 and that the maximum delay for $Treno\_1$ is 1000.

In this example $Treno\_1$ steals the resource from $da0$ because it has an higher priority and the delay it should accept shifting its reservations is too high.

As in the previous example, when $Treno\_1$ sends the QUERY-IF messages, it receives an INFORM from $Nodo\_3$ (in the previous example from $Nodo\_4$, but for the same reason) that specifies that the resource is already occupied by train $da0$, with priority 3, and that the resource will be again available from 310001.

$Treno\_1$ can calculate the total delay it should accept, 3000 milliseconds, that in this case is not acceptable. Having a higher priority than $da0$, $Treno\_1$ can steal the resource: it sends an AGREE message to $Nodo\_3$ and a CONFIRM one to the other nodes, as shown in Figure 13.

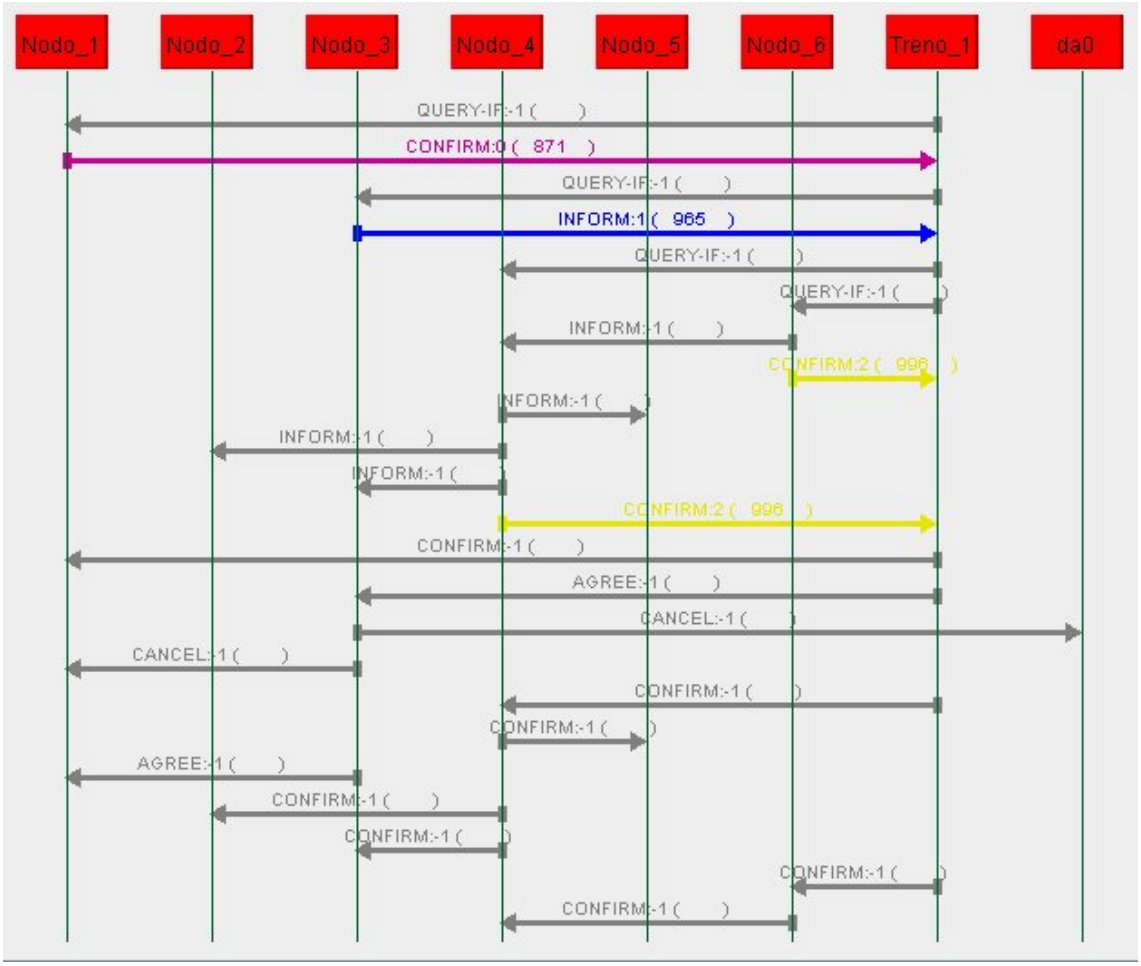


Figure 13: Sniffer view

In Figure 14 the AGREE message sent to $Nodo\_3$ by $Treno\_1$ and the CANCEL message sent to $da0$ by $Nodo\_3$ are reported. In this situation, $da0$ will search for another solution (for example, it will move on $Nodo\_5$), but we do not report this procedure in Figures because it is not interesting in this example. In Figure 15 the complete simulation made with the NetLogo interface is reported. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes 1 second to be completed.

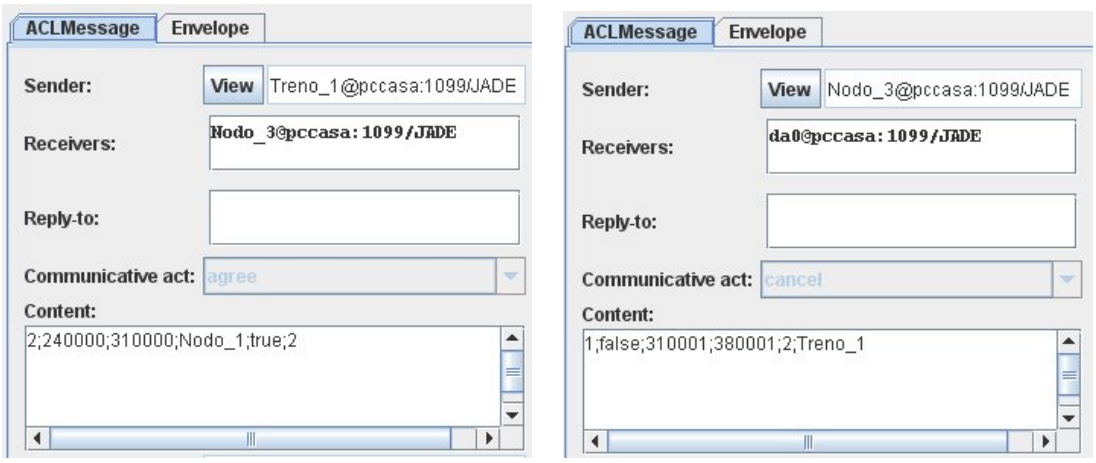


Figure 14: AGREE message sent to $Nodo\_4$ by $Treno\_1$ and the CANCEL message sent to $da0$ by $Nodo\_4$

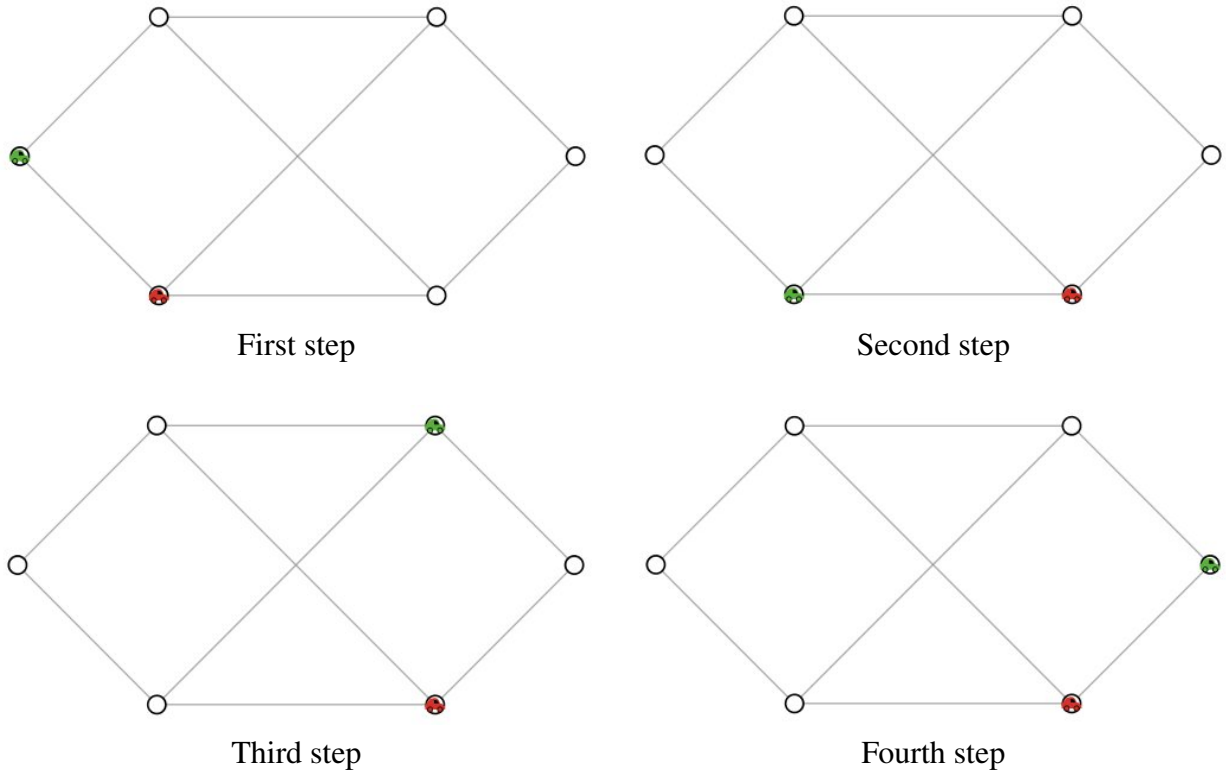


Figure 15: Screen shots from NetLogo

6.5 Path reservation: occupied Node (case 3)

In this example we assume that the dummy agent $da0$, that has already a reservation for the resource $Nodo\_4$ for the interval 310000 - 340000, has priority 1 and that the maximum delay for $Treno\_1$ is 1000.

Being 3000 milliseconds of total delay not acceptable and being the priority of $da0$ higher than the one of $Treno\_1$, the train can only search for another path. It asks the Paths Manager Agent (PA) to get the list of alternative paths (from $Nodo\_1$ to $Nodo\_6$), avoiding $Nodo\_4$. The content of the QUERY-IF message sent by $Treno\_1$ to PA is the next one:

$1; Nodo\_1; Nodo\_6; Nodo\_4; [Nodo\_1, Nodo\_3, Nodo\_4, Nodo\_6]; Nodo\_4$

where the format is the one described in 4.3.2.2, while the content of the INFORM message received from PA is the next one:

$[(1\_false; 2\_true; 5\_true; 6\_false), (1\_false; 3\_true; 5\_true; 6\_false)]$

that is the list of alternative paths (where 1_false means $Nodo\_1$ that is not a "Stop Node"). In this case, $Treno\_1$ selects the path $(1\_false; 2\_true; 5\_true; 6\_false)$ and it succeeds in reserving it, as shown in Figure 16. In Figure 17 the complete simulation made with the NetLogo interface is reported. Starting from the creation of $Treno\_1$, the execution of this example in JADE takes less than 2 seconds to be completed.

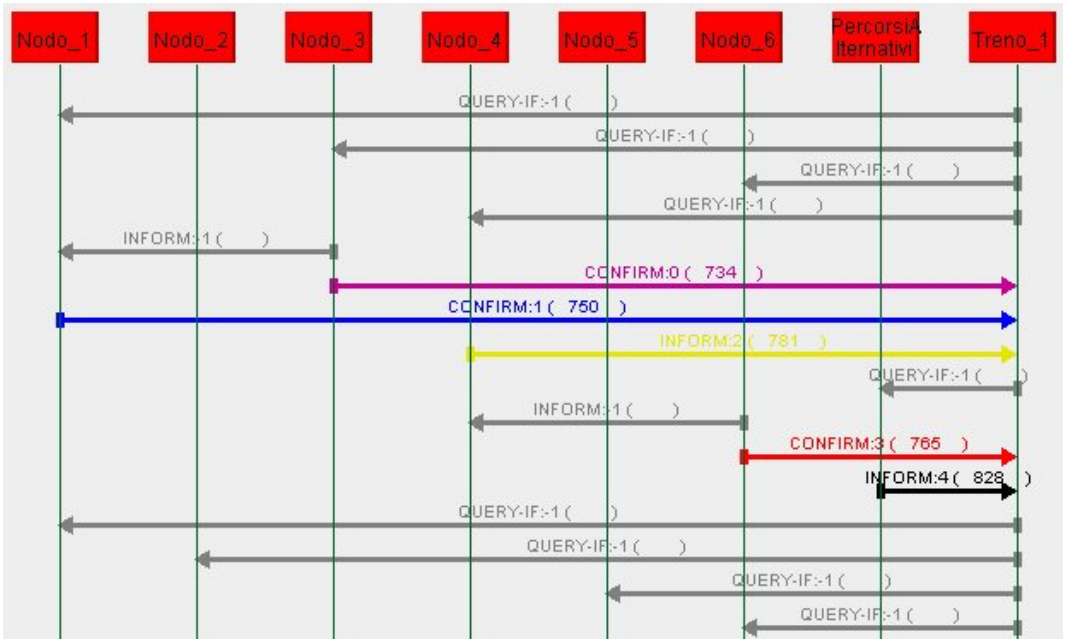


Figure 16: Sniffer view

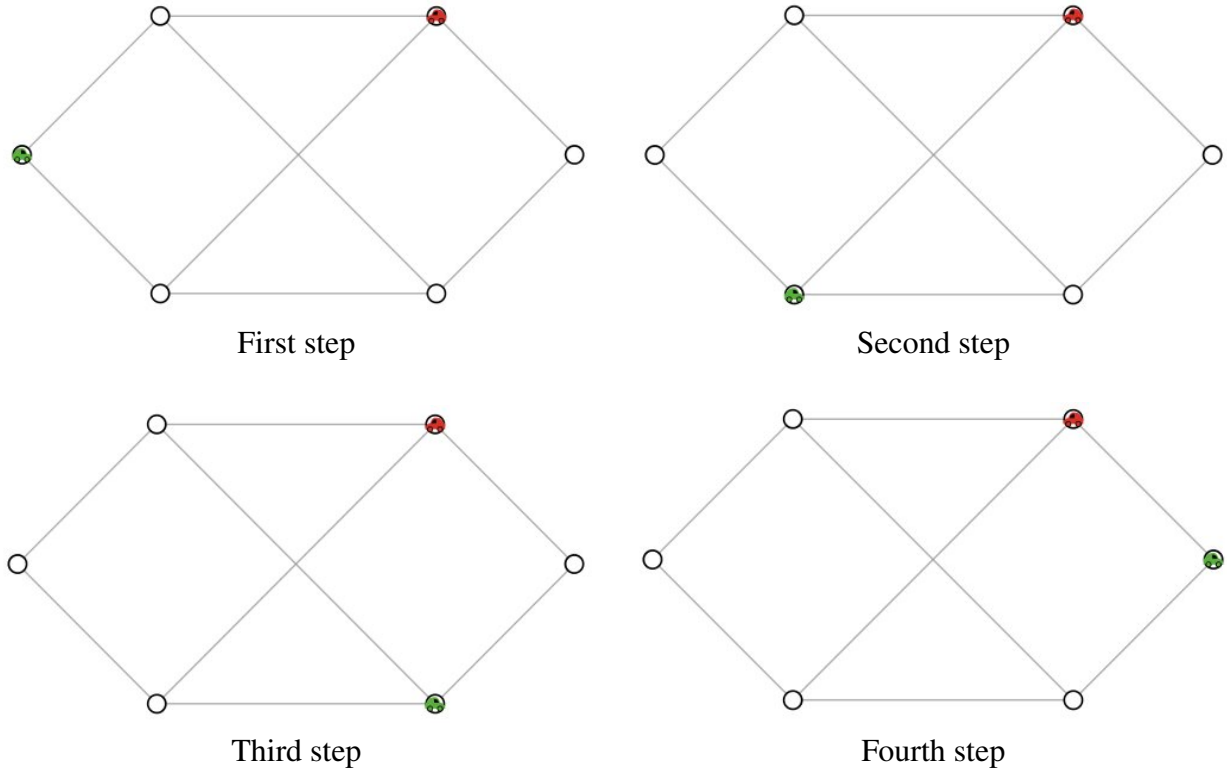


Figure 17: Screen shots from NetLogo

6.6 Out of order Node

In this example we describe the particular situation when the physical resource managed by the Node breaks: the Node is informed of this by "Prenotazioni Stazione" and it sets its status to "out of order", answering Trains that it is not available for a specific period.

This example shows the behavior of the system when a Node goes out of order. In this case only the node goes out of order, while all its arcs are still available.

In Figure 18 we report the structure of the graph while in Table 2 we report the initial plans of two Trains ($Treno\_1$ and $Treno\_2$).

The aim of this example is to show what happens when $Nodo\_4$ goes out of order from 260000 to 300000, while $Treno\_1$ is already on it. $Nodo\_4$ will send a FAILURE message to $Treno\_1$ to inform it that it must stay on the node more than planned (till 300000): the message reports that the reservation number three (that $Treno\_1$ previously in the protocol sent to $Nodo\_4$) must be changed, because the node is out of order ("rottura nodo"), till 300000. At the same time $Nodo\_4$ will also send a CANCEL message to $Treno\_2$, which had a reservation after $Treno\_1$,

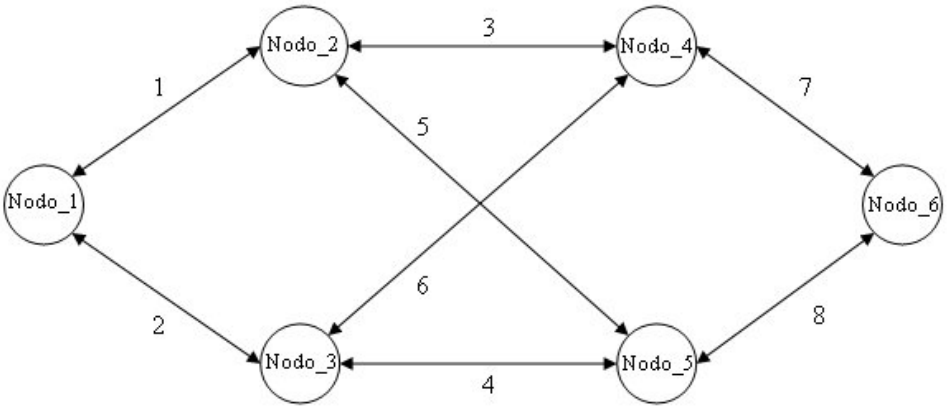


Figure 18: Graph architecture

Train	Step	Node	From (ms)	To (ms)
Treno_1	1	Nodo_1	150000	180000
Treno_2	1	Nodo_1	180000	200000
Treno_1	2	Nodo_2	180000	250000
Treno_2	2	Nodo_3	200000	280000
Treno_1	3	Nodo_4	250000	280000
Treno_2	3	Nodo_4	280000	320000
Treno_1	4	Nodo_6	280000	320000
Treno_2	4	Nodo_6	320000	360000

Table 2: Initial trains plans

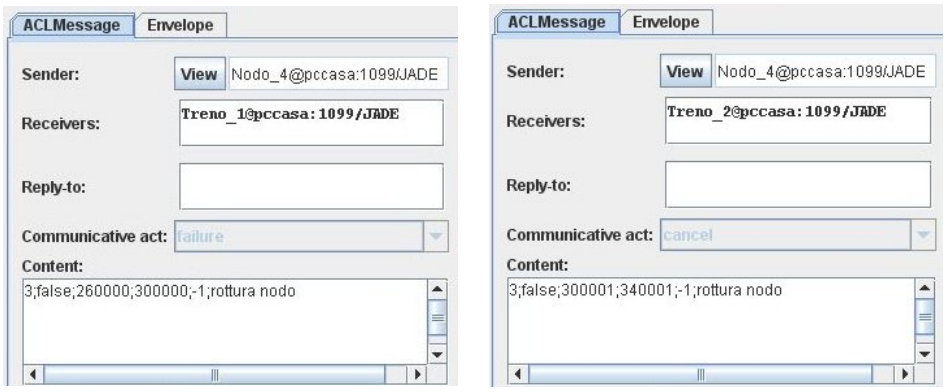


Figure 19: FAILURE and CANCEL messages

specifying that the reservation number 3 (of $Treno\_2$) has been canceled and that the new availability of $Nodo\_4$ will start from 300001, end at 340001, due to its out of order status ("rottura nodo"). In Figure 19 the two messages are shown.

After these notifications $Treno\_1$ will wait more on $Nodo\_4$ and will shift all its reservations, and $Treno\_2$ will adapt its plans and will reserve the new path, updated with respect to the new availability of $Nodo\_4$. In Figure 20 the complete exchange of messages after that $Nodo\_4$ is informed that it is out of order by the Resource Agents Manager is shown, while in Figure 21 the complete execution on NetLogo is reported. The execution of this example on JADE takes about 2 seconds to be completed.

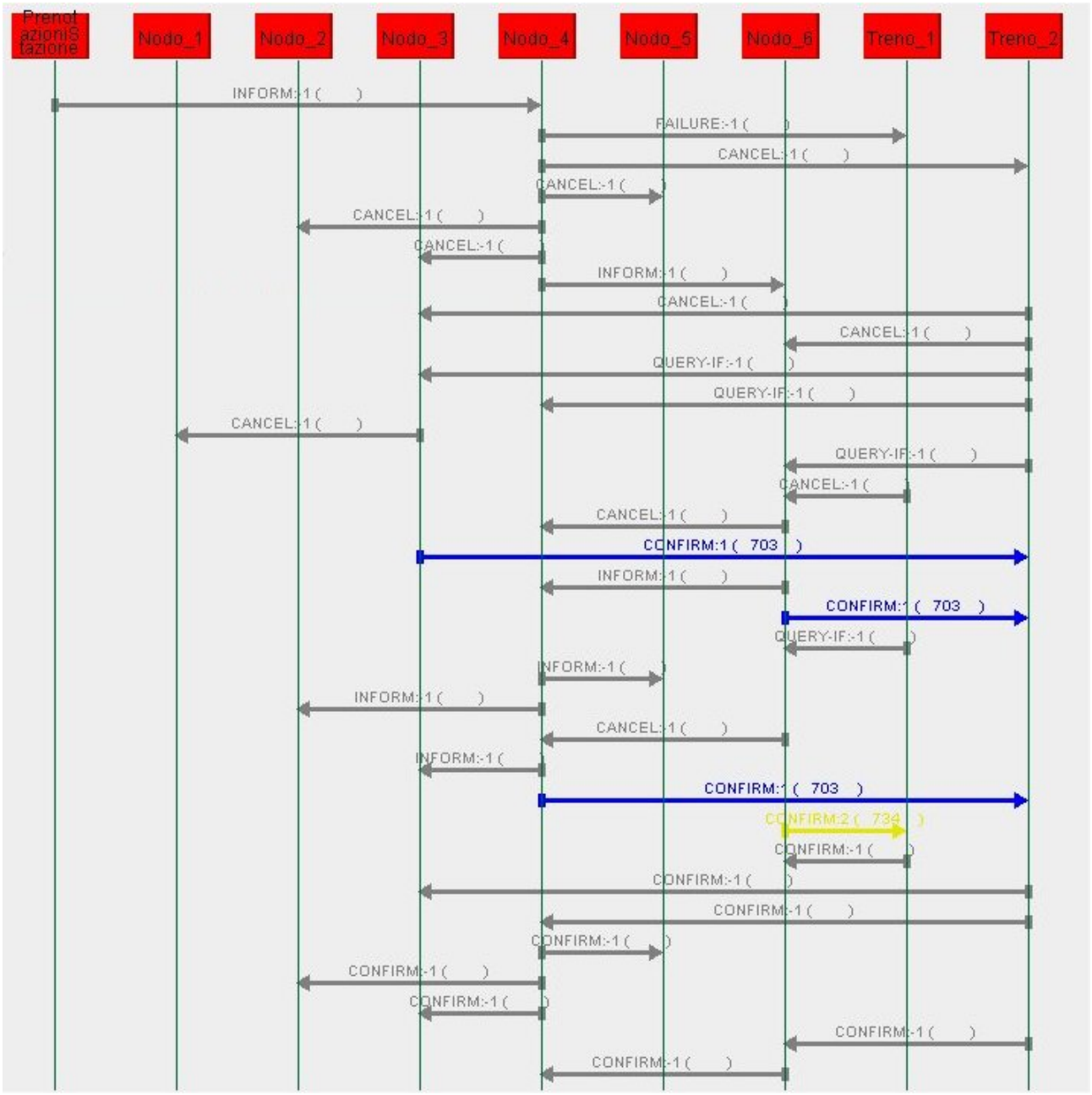
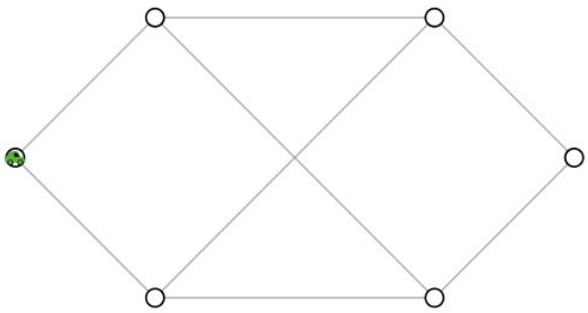
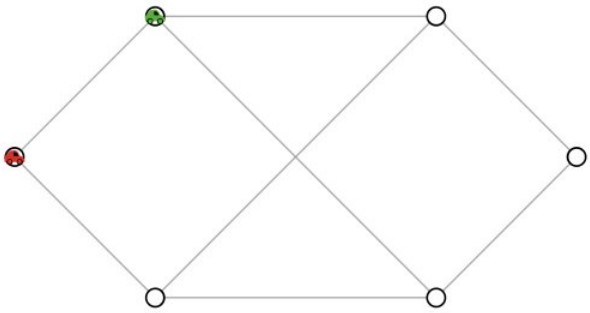


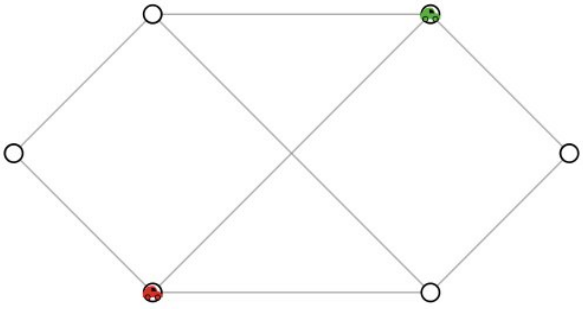
Figure 20: Sniffer view



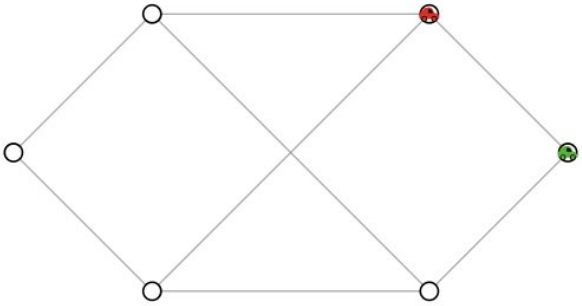
First step



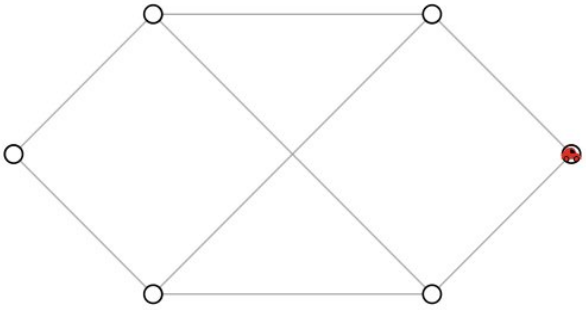
Second step



Third step



Fourth step



Fifth step

Figure 21: Screen shots from NetLogo

6.7 Out of order arc (case 1)

Let us describe a different situation: considering the same station, as reported in Figure 18, and the same two trains (with the original plans described in Table 2), of the above example, suppose that the message received by $Nodo\_4$, coming from "Prenotazioni Stazione", informs that the arc number 6 is out of order from 260000 to 300000, while the node is still available.

In this case, the reservation of $Treno\_1$, that is already arrived on $Nodo\_4$, is not affected, while the reservation of $Treno\_2$ must be canceled, because in the requested interval (280000 - 30000) the arc number 6 is out of order (that is, $Treno\_2$ cannot use it to reach $Nodo\_4$) and because there are no other arcs connecting $Nodo\_3$ with $Nodo\_4$. In the CANCEL message $Nodo\_4$ will inform the train that the node will be available (in this case, reachable) again from 30000. $Treno\_2$ (let us suppose that it can accept 2000 milliseconds of delay) will recalculate the arrival and departure time for each node in the current path, will send the QUERY-IF to the Nodo agents and then it will confirm these new reservations. The complete interaction is reported in Figure 22.

The updated path of $Treno\_2$ is reported in Table 3: note that when the information about the out of order status of arc 6 arrives, $Treno\_2$ is already on $Nodo\_3$, so the updated path will start from that node (and no reservation request will be sent to $Nodo\_1$). Considering the previous example, the systems acts in similar way in these two cases (because we assume that $Treno\_2$ can accept the total delay, so it will wait). The difference is that in this second case $Treno\_1$ is not involved at all. The NetLogo execution is equal to the previous one, in Figure 21, as the execution time on JADE.

Train	**Step**	**Node**	**From (ms)**	**To (ms)**
Treno_2	2	Nodo_3	200000	300001
Treno_2	3	Nodo_4	300001	340001
Treno_2	4	Nodo_6	340001	380001

Table 3: Initial trains plans

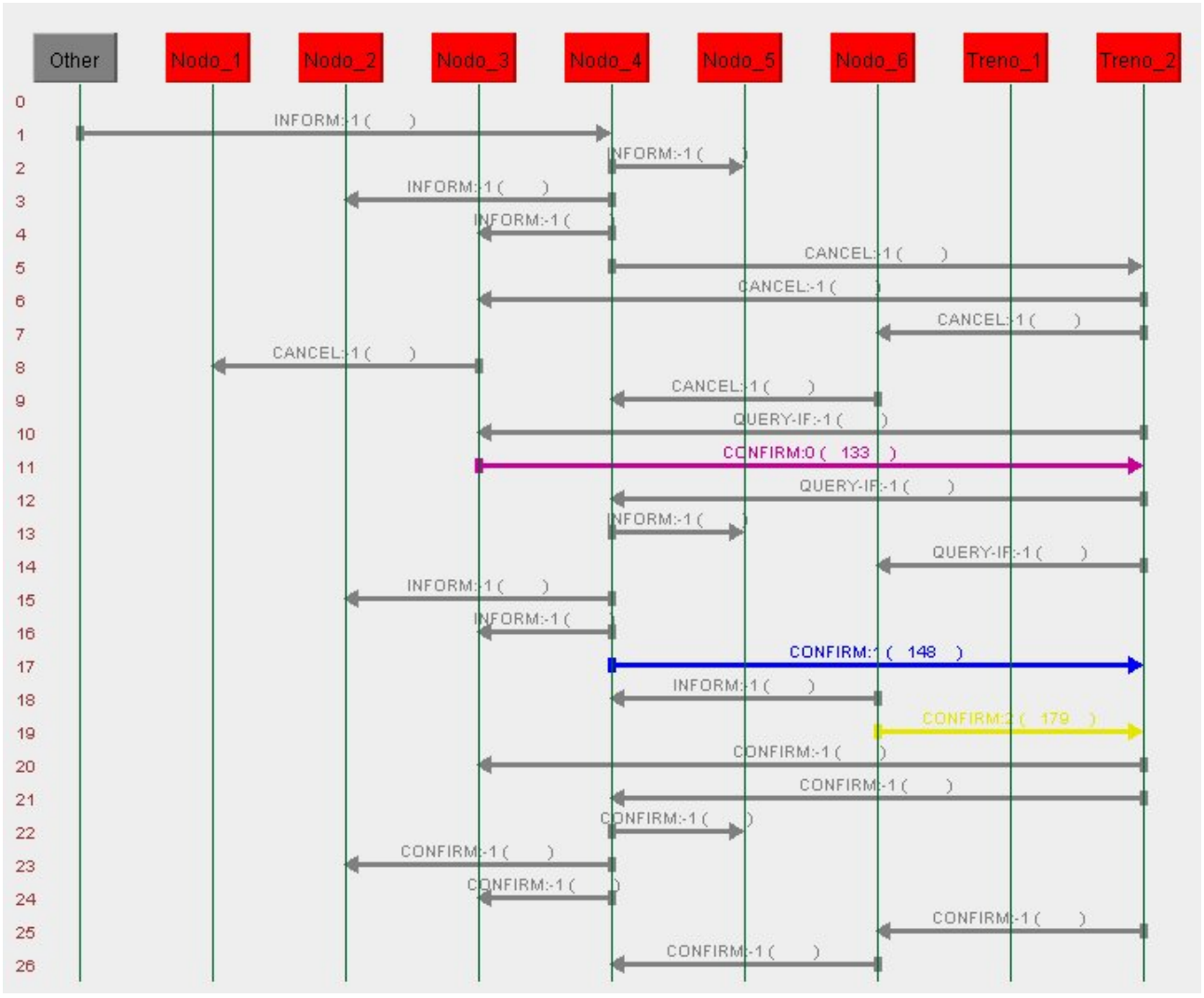


Figure 22: Sniffer view

6.8 Out of order arc (case 2)

To conclude these examples on the out of order status of nodes and arcs, let us suppose that another arc, number 9, connects $Nodo\_3$ with $Nodo\_4$, as shown in Figure 23.

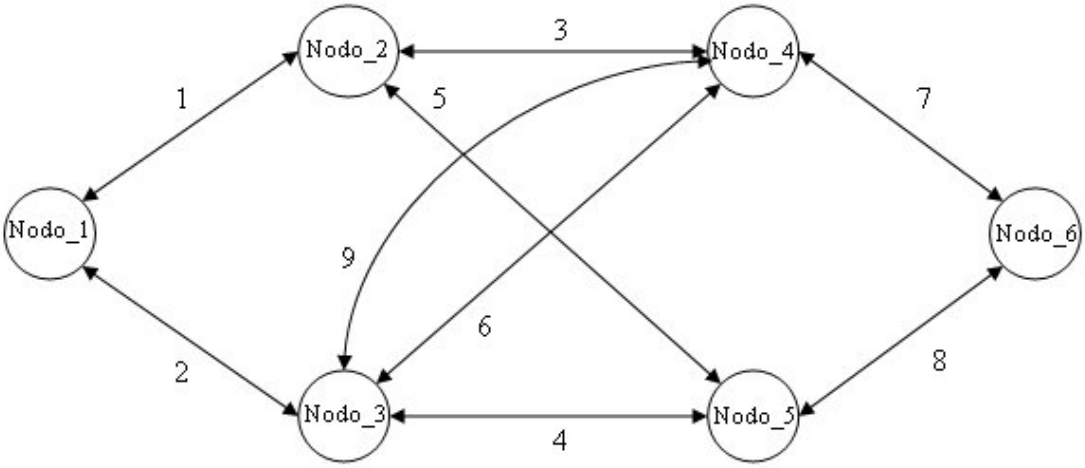


Figure 23: Graph architecture

The original plans of $Treno\_1$ and $Treno\_2$ are always the same (Table 2) and again "Prenotazioni Stazione" informs $Nodo\_4$ that the arc number 6 is out of order from 260000 to 300000, while the node is still available.

In this case, the reservation of $Treno\_1$, that is already arrived on $Nodo\_4$, is not affected, while the reservation of $Treno\_2$ must be modified: in the requested interval (280000 - 30000) the other arc, number 9, connecting $Nodo\_3$ with $Nodo\_4$ is still available and is also free. So $Nodo\_4$ can simply inform (with a PROPAGATE message) $Treno\_2$ that it must use arc 9 to reach it. $Treno\_2$ will update its reservation but does not need to change anything else.

The messages exchanged among agents after the INFORM message from "Prenotazioni Stazione" are reported in Figure 24. The NetLogo execution is equal to the previous one, in Figure 21, and the execution time on JADE is less than 1 second.

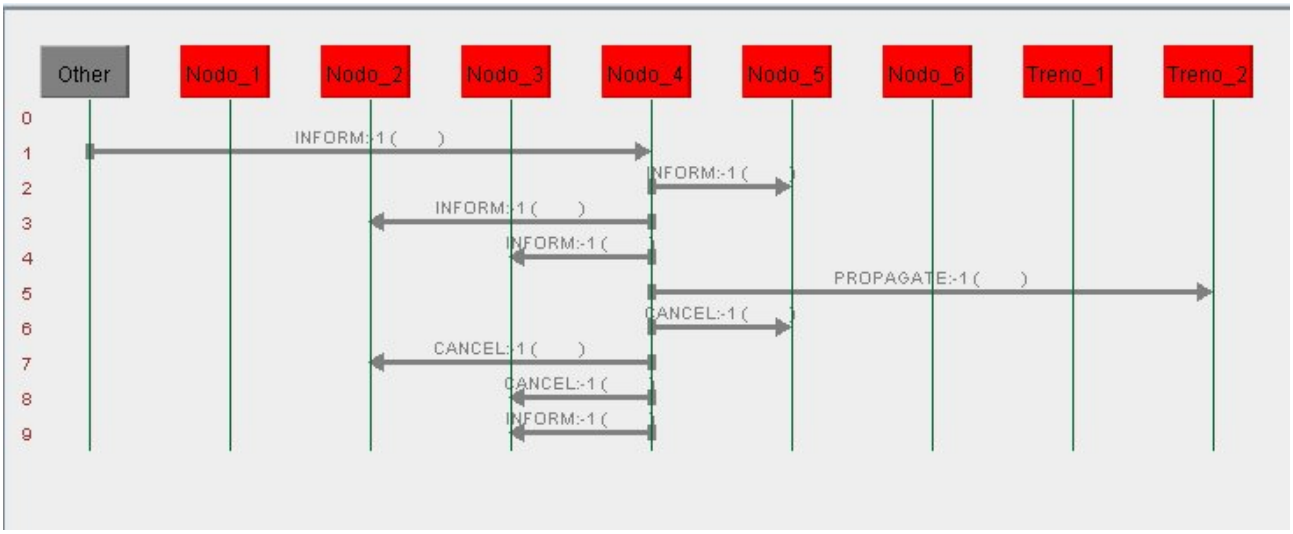


Figure 24: Sniffer view

Chapter 7

State of the art of Resources allocation problems: protocols and comparison

In this chapter we present the results of a deep analysis that we conducted to identify in the literature some protocols, or running systems, that can be compared with ours: due to the large amount of material that we found, we cannot provide a comprehensive survey on the related work and we limit ourselves to analyze those works that are definitely closer to ours either because of the exploited techniques, or because of the domain of interest or because they present a solution similar to FYPA. The chapter describes the selected systems and reports their main features, comparing them with FYPA. Then a detailed comparison among FYPA and the selected protocols, considering some more technical details, is reported.

In particular we focus on those works where an interaction between agents is required to coordinate themselves for moving in a well defined area or for achieving a final goal. Our protocol establishes how agents should interact for finding a sub-optimal allocation of resources distributed over a graph, but similar situations can be found also in other application domains, with different constraints and actors.

In the next sections we present some protocols that are relevant for our work because they manage problems very similar to ours and propose solutions that can be compared with the negotiation protocol described in this thesis. Those protocols deal with problems coming from different domains, but the underlying common point is always the resource allocation problem.

We will summarize:

- MPCA (Multi-Party Collision Avoidance) [126]: airplane collision avoidance
- Airplane rerouting (APR) [15]: airplane collision avoidance
- Waypoints [101]: autonomous robots collision avoidance

- SPAM (Scalable Protocol for Anytime Multi-level) [85]: target tracking with sensors

showing the main aspects of the protocols and their differences/similarities with respect to our protocol. A systematic comparison will be presented in the last Section 7.5.

7.1 MPCA

An area where multiagent systems are very often used is the one of "collision avoidance": in this domain the system reflects a situation where unmanned entities (autonomous airplanes) need to move avoiding crashing, respecting constraints on the path they can use, time to make a decision, distance they need to maintain between them and so on.

Usually, this type of problem is faced in the airspace management, where unmanned little planes need a protocol to negotiate the route, or in software applications where the multiagent system must suggest to the user possible alternative reorganizations of the route of many planes, or often similar problems are found in the management of autonomous robots.

Regarding the airspace management, an accurate research have been made by the "Agent technology center" (http://agents.felk.cvut.cz/) that has developed "AGENTFLY", which is "a multiagent system enabling large-scale simulation of civilian and unmanned air traffic. The system integrates advanced flight path planning, decentralized collision avoidance with highly detailed models of the airplanes and the environment" [1]. The "AGENTFLY" project is still alive and its authors are improving it following many different ways. Its license was sold to BAE Systems as a testbed simulation platform. It is used by US Air Force, IHMS at Florida and several universities. Its authors are working with the Federal Aviation Authority (FAA) using "AGENTFLY" as a tool on their computation grids.

Besides this system, the authors presented many studies on protocols to avoid collisions in airspace.

In [126] two algorithms to avoid collisions among aerial vehicles are presented. The authors were very kind and accepted to review our analysis (in this section and in section 7.5) of their work, so we are able to provide some details also on the AGENTFLY project in Section 7.5, where we add some footnotes to Tables to report additional information submitted by the authors.

In a three dimensions space a group of autonomous airplanes with a mission need to coordinate themselves to avoid collisions: a mission is made of several points that must be reached in a specified time interval. To fulfill the mission, the airplane will follow a list of steps, each characterized by a manoeuvre, a direction, a velocity, starting from the previous step. More, every plane must maintain a minimum distance from all the others, and there are no-flight zones in the airspace. When two, or more, airplanes have a part of the plan in common, they need to change it to avoid crashing.

Every agent can only interact with the ones within a range R defined at the start of the simulation: these are the other planes that it can "see" (on a virtual radar) from its position in the space. Every agent sends to the others it sees an update on its future mission steps so they can check if there are conflicts.

In the paper they propose two solutions: a local one and a global one.

In the local one, the two agents involved in a conflict propose, at first, a list of possible changes to their path and look if they can adopt one of these change to avoid the conflict. If they are not able to find a solution based on the list of generated maneuvers, they generate more maneuvers (accepting higher values for the parameters, for examples trying to move more to left, speeding up/slowing down more, making the altitude higher and so on) and add them to the list: then they try again to solve the conflict. They repeat this procedure until they find a solution. In this way the list can be filled with tens or hundreds maneuvers and the solution is always found. A random choose is used when there are several solutions with equal utility value (using an utility function not described here).

When they have solved their conflict, they will check again if other collisions exist and will start again the protocol to solve the first one.

If an agent is involved in more than one collision, it will solve the one that is expected to occur first in the time.

This algorithm is called "Iterative peer-to-peer collision avoidance (IPPCA)".

In the second algorithm (the one proposing a global solution) called Multi-Party Collision Avoidance (MPCA), authors enlarge the set of agents involved in the collision to find a better solution: in this case the idea is to give the colliding agents enough space to make their evasion maneuvers (to avoid all the agents around) without changing too much their plans. The agents that have a colliding path create a group: then they try to change their paths (they create a list with all the possible modifications to their paths) and add to the group those agents that could be interested by these path changes. So the group is enlarged to involve all the agents that are near to the ones that have a conflict. Then, the possible alternative plans are analyzed by the group till a solution is found, and all the agents will modify their path as decided.

The group searches the states space of possible plan changes (due to all the possible applications of evasion maneuvers sequences, velocity change and so on) using an A* algorithm modified for the situation.

To simplify the communication and synchronization issues the authors have implemented this protocol creating, for each group, a coordinator agent that collects all the information it needs to solve the problem and than to find a solution. In this way the organization of the algorithm is partially centralized.

If an agent is involved in more multi-party groups, it will only join the one with the earliest

expected collision in the time line.

As the authors kindly explained us, MPCA could be used as a global protocol, but it is not intended to be so. The idea is to used it as a "local/global" protocol. This means to compute "globally" a solution for a small group of airplanes, that are involved in a single collision situation, but in the meanwhile another group of agents can solve different collision in their corner of the world. So MPCA was designed to be something between local and global: it is used to solve "globally" ad hoc local problems.

The MPCA has been tested using AGENTFLY and the comparison has been made using the IPPCA algorithm, implemented as plug-ins in AGENTFLY.

The observed differences between IPPCA and MPCA are:

- message flow characteristics: IPPCA has quite steady bandwidth of communication flow. MPCA has high peaks and then very low communication (as the data are computed by the coordinator);
- quality of solution: MPCA provides better solutions because the A* is able to find solutions that are not checked with IPPCA;
- computation demands: as the MPCA goes through much bigger space, the computation is much longer. This one of the reasons to keep MPCA local by restricted size of the group.

The MPCA algorithm in particular is similar to ours, because there is the idea of "moving others to get space for you". The domain instead is quite different, because the airplanes have much more flexibility in their movements (they must avoid certain zones and avoid other agents, but must neither follow rigid and limited paths, nor have to reach a fixed point but and area) and, above all, the move in a 3D space.

7.2 Airplane rerouting (APR)

Another attempt to apply Multiagent system to this problem comes from Agogino and Tumer in [15]. In their work the authors present a multiagent structure to control air traffic flow using tree kinds of "change mechanisms", explained later, and then analyze a learning algorithm to improve the system efficiency (that we do not report here because it is out of our scope). The domain is the one of US airspace, where the space is divided into regional centers and again into sectors. The algorithm uses a global evaluation function that considers the congestion in a particular set of sectors and the global air traffic delay. Using this common function agents independently take decisions about how changing their plans. In this system agents are ground location throughout the airspace and are called "fixes". Each agent is responsible for the aircrafts going through its

fix. Every airplane has a "flight plan" consisting of a sequence of fixes. In this organization agents can change the plan of the interested airplanes in three ways:

- Miles in trail (MIT): agents control the distance that the airplane must keep from each other while approaching a fix. If the MIT values is high, fewer planes will be able to cross this area because they need to slow down their velocity to maintain the distance
- Ground delays: an agent can control how long aircrafts that will eventually go through a fix should wait on the ground, that is, the airplanes will arrive later at the fix
- Rerouting: an agent can divert the foreseen planes of its fix making them choosing another path.

The algorithm identifies sets of agents that can influence themselves rerouting airplanes in their fixes: each agent lists the possible solutions to the congestion problem and then chooses the best solution using different learning algorithms (the authors study some types of learning strategies and compare them).

We do not understand so clearly (and were not able to contact the authors) whether the agents can use in the same simulation all the three techniques seen above to change the plan (or if they must use only one type in a run) and furthermore we did not find the strategy used to choose among these strategies (if they are foreseen simultaneously).

The implementation of the algorithm has been developed using FACET (Future ATM Concepts Evaluation Tool) [21], that receives scripts from the agents and simulates the execution of the algorithm and gives them back the overall impact of their changes to the airplane's plans.

This algorithm is interesting but seems to work only if the groups are limited to few agents, because it is based on the list of possible choices, that becomes too long if there are large sets of agents, or too many possibilities of plan changes. More, the agents seem not to negotiate, but only to work independently using the same evaluation strategy and only one possible change to the airplane route. All these constraints seem to limit the applicability of the algorithm to very complex scenarios or short term simulations, as we needed.

7.3 Waypoints

Similar problems and solutions can be found in the field of autonomous robots which are able to independently move and need to avoid collisions while trying to reach a desired destination. This is a common situation for example in the military area, where the robots are entities that must explore territories, or in industrial applications where little robots could be used in places that are not suitable for humans (under-water or mines exploration and so on).

In the area of autonomous robots it is worth mentioning the proposal made by Purwin, D'Andrea and Lee [101]: the authors present a cooperative decentralized path-planning algorithm for a group of autonomous agents that provides guaranteed collision free trajectories in real-time. The algorithm is based on the idea that every agent reserves an exclusive area (called A area) for itself and always remains inside that area, and no two reserved areas are allowed to intersect at any time.

The core of the algorithm is based on pair-wise conflict resolution among two agents. Both agents operate according to exactly the same rules, with the only exception being how priority is assigned. For the algorithm the following assumptions are made:

- Decentralized agents: all computation/control is done on board
- Total number of agents is known
- Motion primitives are available to move the agents in a deterministic fashion
- Point-to-point communication between agents
- Agents can localize themselves, but not others.

The algorithm is executed considering discrete time (agents work with "frames", that is, they assume a discrete divisions of time).

Every agent is trying to reach its task location, which has been selected by some higher level entity that is not in the scope of that work. Position and velocity of the agent are expressed in a global Cartesian coordinate system. The agent's motion is controlled by a deterministic motion primitive MP(), which contains trajectory generation and low-level control of the mechanical actuators (also this aspect is out of the scope of that research). Upon specification of a desired destination D the motion primitive will compute a path that takes the agent to D with zero final velocity.

Every agent stores information about itself and the other agents and is able to communicate using a wireless network.

The algorithm grants that every agent will always, and only, move inside its reserved area, A. Instead of choosing the reserved area (A) directly, agents are indicating their intentions by requesting an area (called B area) first, and exchanging it with the others. Agents can change B arbitrarily. However, significant changes to B can cause the negotiation cycle to start over. The requested area B has to contain the reserved area A at all times, hence, an agent cannot move to a location that is not inside the requested area.

Two intersecting B areas indicate a possible conflict. In this case the agents will negotiate to find out which one gets priority and how the A areas are being selected. The base for this negotiation

is a scalar cost function: the agents with a conflict exchange their respective costs and choose who has the higher priority using this, choosing also who will change its A area. The agent with the lower priority will reduce its A area (stopping and waiting for the other agent to pass over).

With this solution an agent should stop and wait several times, getting a huge delay: the algorithm proposes also a second way of acting. Instead of stopping, an agent can decide to change its path to avoid an obstacle (or simply a point of its paths that intersects with many other agents) and to move around that. It will choose a "waypoint", that is a new intermediate destination, and will try to reach it using the same algorithm shown above. Then it will start again to reach the initial destination.

In the basic algorithm all agents can communicate with all the others: this implementation is foreseen for autonomous robots, with limited battery autonomy and wireless communication limitation, so if the total number of agents is huge then the robots can limit their communication range to reach only the nearest agents, saving energy. This algorithm's limitation is realistic because probably only the agents that are close to each other can have intersecting paths, consequently an agent must inform only its neighbors of the changes to A and B areas.

We do not describe the algorithm with more details because it presents a complex and out-of-our-scope description of the primitives chosen for changing the reserved and requested area, that are managed by a geometrical function and not by the negotiation protocol: the A and B areas will be always rectangles, chosen to approximate the new path. Moreover, the paper presents the pseudo-code of the protocol, so reporting it here is not interesting, and the code in C++ is also available on the author's web site.

This algorithm is similar to ours in the idea of how the agents collaborate to solve the conflicts and how they can change their strategy: in both protocols agents can stop and wait for the other to move, or can change their path.

The difference is that in our domain the paths are limited to a predefined set and are divided into fixed parts. Every subpart is managed by a Resource agent, whereas in that article agents are able to move without limitations and without intermediate agents. More, those agents operate in a wireless environment, so the number of exchanged messages and the real distance of the agents can make the difference on the behavior of the entities: in this case some limitations and heuristics must be adopted (not described here), while in our algorithm the only limitation is due to the computational time of managing all the messages. Anyway we tested our system also with complex configurations (see Section 5.6) and the number of messages does not make the performances degrade. If we wanted to map that algorithm to our domain we should change it in many aspects, and probably these modifications would be too complex to be implemented. We should:

- make the space discrete and bounded to predefined paths
- limit the possible changes of A and B areas to choose some of the possible predefined

paths

- force the agents to cross predefined points along a path
- modify the utility function.

7.4 SPAM

In [85] a "cooperative negotiation protocol that solves a distributed resource allocation problem while conforming to soft real-time constraints in a dynamic environment" called SPAM (Scalable Protocol for Anytime Multi-level) is presented by Mailler, Lesser and Horling. In particular that proposal models the resource allocation problem as a constraint satisfaction problem.

As in the other domains described above, in that work we can find a set of limited resources and some agents interested in using them in different moments for different periods: the resources are three sensors platforms, and agents need at least three sensors to track an entity moving in the environment. More sensors give a more accurate target's location.

In the proposed solution each platform is managed by an agent, which is also in charge of localizing and following a target (this task allocation is made out of the system), deciding which sensors (of which platform) it needs to do this and when. So if two, or more, agents (called also track managers) need the same sensor for different tasks then a conflict arises and it must be solved. The solution is centralized because the agent that first creates the conflict becomes the "mediator" of it and must solve it, asking the other involved agents information and then propagating its decision. Every agent has an utility function U that uses to evaluate the proposed solution (the set of sensors and the period of usage), and can change U to solve quickly a conflict.

SPAM protocol works in two main phases, trying at first to find a solution which avoids the negotiation. At the end of the first stage there is always a solution, even if it is not the optimal one or if it generates conflicts. The algorithm foresees also to lose a target, solution that anyway causes a huge penalization of the social utility.

Stage 1 of SPAM serves two primary functions. The first one is to try to find a solution within the context of the information that the protocol has when it starts up. However, since the protocol attempts to maximize the social utility, each of the agents tries to maximize its local utility without causing new constraint violations. If this can be done, then no further negotiation is necessary, and the protocol terminates at the end of stage 1.

Moreover, in this stage the agents use a "concession rate" to decide if they have to activate the second phase, more expensive, or not: this rate is a percentage of the agent's utility and specifies how much the agent will concede before skipping to the second phase.

The second function of stage 1 is to ensure that some utility is obtained while waiting for stage

2 to complete. If the reason the protocol was started was a resource requirement change, a temporary solution is applied to the problem: this solution, although not conflicts free, has the ability to obtain at least some utility while the mediator tries to get a better solution. Conflicts that are unresolved are actually left to the individual sensor agents to handle.

If stage 1 was activated because of a newly discovered conflict, and a conflict-free solution cannot be found, then the manager just enters stage 2: in this case it does not concede, does not bind a temporary solution, and it does not reset its objective level, but it only enters stage 2 to find a solution.

Stage 2 attempts to solve all local conflicts that a track manager has by elevating the negotiation to the track managers that are in direct conflict over the desired resources. The originating track manager takes the role of the negotiation mediator and starts collecting all the information it needs to generate alternative solutions. These solutions are generated without a global vision, so they are conflict free only from the point of view of the mediator. What this means is that the view of the mediating manager is limited to only the constraints that arise from the sharing of a resource with it. If the solution that will be imposed by the mediator will cause other conflicts then other agents will try to solve them, even starting again all the algorithm.

When an agent is the mediator of a conflict, it starts asking the other agents for meta level information, than it elaborates them and generates all the possible alternative solutions, including those where one, or more agents, must lower its utility function. Then it will send the list with the alternative solutions to the agents involved and will wait for their responses: the other agents will reorder the list using their local information and utility function and will send back this reordered list to the mediator. Last, the mediator chooses a solution, possibly a good one for all the agents involved, and sends it to the agents that must apply it.

At this point, each of the track managers is free to propagate and mediate a new negotiation if it chooses to enter the second stage. At the time when this article was written, SPAM allows the agents to enter potential oscillations, maintaining no prior state other than objective levels, from negotiation to negotiation and rely on the environment to break oscillations.

To test the SPAM protocol the authors implemented a model of the domain in a simulation environment called Farm [66], a component-based distributed simulation environment written in Java. They performed tests to evaluate the performance of SPAM compared with those of a Greedy Tracking Agent and an Optimal Tracking Agent. To do this, the simulation used not moving targets.

In the simulation with moving targets (a not comparing test, only a performance test), SPAM gives good results, near to the optimal, and shows a linear increase in the time needed to converge when the problem gets harder, but they do not have implemented other solutions to compare with.

Comparing this protocol to ours, the main difference seems to be in the possibility of loosing a track: in this algorithm this event is allowed, even if it is the last solution, but in our protocol

this is not possible, because it means a physical incompatibility. Furthermore, the representation of the domain as a constraint problem is more difficult for our domain, and last we preferred a real distributed negotiation, while the solution proposed in [85] is partially centralized. The agents do not negotiate, they choose one of them to solve the constraint problem and then apply its solution.

7.5 Comparison

In order to draw a systematic comparison between the works discussed in this chapter and ours (named FYPA in the next Tables), we identified a set of features relevant for characterizing a negotiation protocol.

7.5.1 Accepted agent definition

For sake of clarity, we recall the definition of "agent" given in Chapter 2 (from Jennings, Sycara and Wooldridge [73]), to which we adhere:

"An agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives. There are thus three key concepts in our definition: situatedness, autonomy, and flexibility. [...] By flexible, we mean that the system is:

- responsive: agents should perceive their environment and respond in a timely fashion to changes that occur in it;
- pro-active: agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate;
- social: agents should be able to interact, when appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities."

Hence, the first characterizing feature we consider in our comparison is what definition of agent is accepted by the authors, specifying which standard features can be found in the MAS discussed above.

- **Are agents autonomous?**
- **Are agents situated?**

- **Are agents responsive?**
- **Are agents pro-active?**
- **Are agents social?**

Table 1 provides a comparison among the protocols we described in this chapter, with respect to the accepted definition of agenthood. For every parameter the possible values are:

- YES: the agents show all the typical features associated with the parameter
- NO: the agents show none, or very few, of the typical features associated with the parameter
- LIMITED: the agents show some of the typical main features associated with the parameter

In particular, "Limited" referred to sociality underlies a solution where the agents communicate among each others but use very simple data structures and, above all, this action is a simple exchange of information: the solution to the problem is partially centralized, so the agents are more "communicative" then "social". The same value for the "Pro-activity"[1] parameter underlines that the main behavior of the agents is passive, that is, they wait some changes in the environment happen before acting. So these agents are more reactive than proactive.

	Autonomy	**Situatedness**	**Reactivity**	**Pro-activity**	**Sociality**
MPCA [126]	YES	YES	YES	NO	YES
APR [15]	YES	YES	YES	LIMITED	LIMITED
Waypoint [101]	YES	YES	YES	NO	LIMITED
SPAM [85]	YES	YES	YES	NO	LIMITED
FYPA	LIMITED	YES	YES	YES	YES

Table 1: Comparison: accepted agent definition

[1]Concerning the MPCA protocol, the value NO for the filed "Pro-activity" is correct if we limit our evaluation only to the MPCA protocol, but if we consider agents in the whole AGENTFLY project (those called "pilot agent"), they are pro-active as well. At the beginning, every pilot agent receives its mission (which can be changed by human operator during the simulation) and the agent plans its trajectory to fulfill its mission or to communicate with other agents if it is in a group mission (see "tactical-agentfly" at http://agents.felk.cvut.cz/)

7.5.2 Domain, Purpose, Approach of the MAS

Table 2 provides a comparison with respect to the general features of the protocols.

- **Which is the domain where the protocol is applied?** The described protocols refer to different applicative domains, that are summarized in this way: Railway management ("RAIL"), Air traffic management ("AIR"), Sensors management ("SENSOR") and Autonomous robots's path management ("ROBOT").
- **Is the protocol used for simulation purposes (for example, for performing a what-if analysis or for implementing a decision support system)?** For this parameter the value will be "YES" if the system/protocol has been implemented/used in simulations too
- **Is the protocol used for controlling agents (hence, not for simulation purposes, but for allowing real agents to negotiate)?** In this case, the value will be "YES" if, and only if, the system has the real control over physical entities, that is, its decisions are not checked by an human operator. The value will be "Limited" if a human user will accept the proposed solutions before applying them.
- **Which is the approach underlying the protocol?** For this parameter we list the main standard techniques used in the protocol, for example Game theory, Auctions and so on.
- **Whatever the purpose (simulation or control), domain and approach, is the system used for a real industrial application? Which one?**

7.5.3 Analysis and design of the MAS negotiation protocol

Table 3 shows the values for the parameters that we considered for the design features of the protocols.

- **Which is the detail level of the MAS design?** The possible values are: High (Verbal description of the system), Low (Detailed description with use cases or Class Diagram or other languages)
- **Is the pseudo-code, or a simplified version of the code, of the negotiation protocol available?**
- **How many different agent roles does the MAS include?** The number refers to the different roles a single agent is able to assume

	Domain	Simulation	Real control	Approach	Industrial
MPCA [126]	AIR	YES	YES	A* state search	YES (BAE Systems and others)
APR [15]	AIR	YES	LIMITED	Learning algorithm	LIMITED (foreseen for the Federal Flight Administration)
Waypoint [101]	ROBOT	NO	YES	Scalar cost function, Computational Geometry	NO
SPAM [85]	SENSOR	YES	NO	Distributed Constraints Satisfaction Problem	NO
FYPA	RAIL	YES	LIMITED	Distributed resources allocation	YES (Ansaldo STS)

Table 2: Comparison: Domain and simulation

- **How many different kinds of messages do the agents exchange?** The number indicates the type of different ACL messages (if known) or different semantics (request, answer, update...)
- **Is the solution computed in a partially centralized way (at least, in any iteration step of the negotiation)?** The value will be YES if the final solution in calculated by only one agent[2], NO if the solution emerges from a real negotiation among agents
- **Is the algorithm guaranteed to terminate? Under which conditions?** The value will be NO if the algorithm is allowed to enter a loop or a situation where it does not assure to find a solution within a specified time, while the value will be YES if the algorithm will always find a solution. In this case, if the solution is partial or it accepts some constraints violation, this will be specified between round brackets.

If we were not able to clearly understand the correct value for some parameter from the papers we report in this Table (and in the next one) the value UNK, that stands for "unknown".

[2]Concerning the MPCA protocol, a decentralized implementation exists but it is not published, so we do not consider it in this thesis.

	Design	Pseudo-Code	Roles	Messages	Centralized	Termination
MPCA [126]	HIGH	YES	2	2	YES	YES
APR [15]	UNK	NO	1	2	NO	YES
Waypoint [101]	UNK	YES	1	3	NO	YES
SPAM [85]	UNK	NO	2	4	YES	YES (but with no assurance that all tracks have been managed)
FYPA	Low	YES	3	7	NO	YES (a no-way-out situation is reported to the user)

Table 3: Comparison over the design features

7.5.4 Implementation of the MAS negotiation protocol

In Table 4 we summarize the values for the parameters regarding the implementation features of the protocols.

- **Is the MAS implemented?**
- **In which programming language?**
- **Is the MAS based upon an existing agent platform?**
- **Can the strategy of the agent vary during the same execution run?** With "varying strategy agent"[3] we intend an agent that is able to act in different ways using different rules. That is, it is able to find many ways to solve a problem during the execution/simulation
- **Is the code implementing the protocol available? Under which license?** The possible values are: "FREE" if the code is available under an open license, "NO" if the code is protected by a Non disclosure agreement or it has been registered for a third private entity or "UNK" if we do not know how/it the code is available.
- **Can the user interact with the MAS on-line during the MAS execution?** If the value is "YES" it means that the user can change the execution of the system while it is running, using a GUI or other techniques.

[3]Concerning the MPCA protocol, once the MPCA or IPPCA is used for solving particular problem, the agent will use it until a solution is find or timeout will pass. Considering the global "AGENTFLY" project, pilots agent have several different methods to solve collision avoidance (MPCA and IPPCA are just some of them) and they select the best method based on current situation (time to solution, mission goals, environment and so on).

- **Which GUI is available?** The value can be "ON LINE" if a GUI exists and the user is allowed to modifies the execution of the system, "OFF LINE" if something exists that shows the user the execution of the protocol (during the execution or later), "NO" if the system only gives the result but is not able to let the user understand the intermediate steps of the protocol. The value will be "PH" if the system is physically implemented so its execution is visible for the user (you can see for example the physical entities moving) but it can not be considered a GUI.

	Implemented	**Languages**	**Platforms**	**Var. Strategy**	**Code available**	**User**	**GUI**
MPCA [126]	YES	Java	AGENTFLY/ AGLOBE [12]	NO	YES (for academic purposes only)	YES	ON LINE
APR [15]	YES	UNK	FACET	NO	UNK	NO	NO
Waypoint [101]	YES	C++	2005 Cornell RoboCup	YES	FREE	NO	PH
SPAM [85]	YES	Java	FARM	YES	UNK	NO	NO
FYPA	YES	Java	JADE	YES	NO	YES (JADE interface)	OFF LINE

Table 4: Comparison over the implementation features

7.6 A comparison with the Contract Net Protocol

The Contract Net Protocol (CNP in the next) [111] is a well known standard protocol used to allocate tasks in a MAS where one, or more, agents called "initiator" must complete a task and other agents, called "participants", can execute this task. The aim of the CNP is to find an allocation of tasks to participants. We know that the CNP is based on an auction where the initiator sends a "call for proposal" to all the participants and the participants answer, if they can do the job, specifying the cost of executing the task. Then the initiator will choose the best participant (the one offering the lowest price) and will inform the chosen participant that it has won the auction (while it will inform the others that they have lost). This procedure is repeated for every initiator, that is, if the task to be done is the same, the participants will be involved in many auctions.

Changes to the CNP have been proposed in many studies, to avoid for example the situation when a participant is not able to make a proposal for a task because it is waiting for the response of another auction. A solution to this problem is proposed in [74], where an extension of the CNP called CNCP (Contract Net with Confirmation protocol) is described to find a better allocation of tasks to the participants.

While analyzing the FYPA problem we considered this protocol, but there is an important limitation at the lowest level that convinced us to not use the CNP: if we are modeling FYPA as a CNP, we should identify the initiators and the participants and one or more tasks to be done. In our case, the most natural representation of the problem could be:

- Initiator: Train
- Participant: Node
- Task: to get a reservation of a resource

but this organization is quite strange for a CNP, because we have only one participant for each task. In FYPA no agent has the complete knowledge of the station and every node is managed by one Node agent, that is, when a train needs to use a resource it has to ask only the Node agent managing that resource. There are no other agents (participants) able to execute the same task (to reserve that specific resource). In this situation, to make a "call for proposal" to reserve a resource seems quite useless, because there will be only one participant that can answer. Or, otherwise, what we actually do in FYPA, that is, the direct exchange of messages between the Train and the Nodo agent, can be seen as a step of a CNP: in this case we could say that FYPA is based on a really simplified/strange type of CNP.

Another idea to manage FYPA as a CNP could be the one where the task is "to cross the station using a specific path (Let it be R1, R2... Rn) with specific time intervals". In this case, the call for proposal is sent to all the Nodo agents that should be able to answer something like:

1. ok, the resource Ri (one of those requested) is free (the cost is 0)
2. the resource Ri (one of those requested) will be free from Tx (the cost is the difference in milliseconds)
3. I'm managing a resource that is "an alternative" to Ri (one of those requested) and I'm free when requested

but in this case we must suppose that every agent is aware of being "an alternative" of some node, that is, considering R1 and R2 connected by an arc, R3 is an alternative of R1 and R2 if the arcs R1 - R3 and R3 - R2 exist, and this knowledge is not so simple to extract from the database. Moreover, the agents should also exchange many more information each others to

maintain an updated view of the graph (they should inform also all the alternative nodes, that are many more than the neighbors) and last, but not least, for every call for proposal the answers from the participants will be many, while perhaps the train will use only a subset of them.

This solution should be analyzed more deeply to understand the real complexity but anyway we considered it too complex with respect to the one we chosen with FYPA and without real benefits. Moreover, the first CNP solution could be not considered a real instance of the CNP, so for these reasons we decided to not use the CNP as starting point for our protocol but we designed it "ad hoc".

Chapter 8

Integrating ontologies in MAS

In the previous chapters a detailed description of our negotiation protocols, MAD and FYPA, has been presented, and also a description of the state of the art regarding the resources allocation protocols has been reported. Imaging the possible extensions to both the systems, we considered the integration of ontologies to get more expressive Multiagent systems. The possible enhancement of MAD and FYPA systems using ontologies are described in the end of the chapter.

In this chapter we report the detailed state-of-the-art analysis of the integration of ontologies in MASs that we conducted to understand if, and how, this technology can be exploited in a generic MAS. Our proposal to integrate ontologies in MAS is described too. Later on the state-of-art of the integration of ontologies in Multiagent systems will be presented: we will focus above all on extensions of already existing FIPA-compliant frameworks to integrate the support for the ontologies, looking at the FIPA-standard, and we will then describe also some non FIPA compliant systems, which are interesting different solutions to this integration problem. Furthermore, we will also present our proposal to integrate ontologies in MASs.

In the literature there have been few attempts to realize the FIPA Ontology Agent, and each one adopts a particular point of view of the problem.

Some solutions implement only some services of the OA, others change the FIPA specification of the meta-ontology in order to use the OWL specification language, others realize a Web Service with this role. But besides being different from the design point of view, these solutions are also different in the choice of the framework, and hence of the language used to implement it.

In the next sections we will present the most relevant solutions for the integration of the ontology agent in a FIPA compliant framework, and will discuss differences from the standard, as well as pros and cons of the proposed works.

8.1 Implementation over COMTEC platform

The first attempt to realize an OA was made in 2001 by Suguri, Kodama, Miyazaki [115]. They realized an Ontology Agent for the COMTEC platform: unfortunately, this platform is no longer maintained nor accessible via web.

The implementation of the Ontology Service specification is pretty straightforward based on the specification. Figure 1 shows the overall view of the internal composition of the ontology agent.

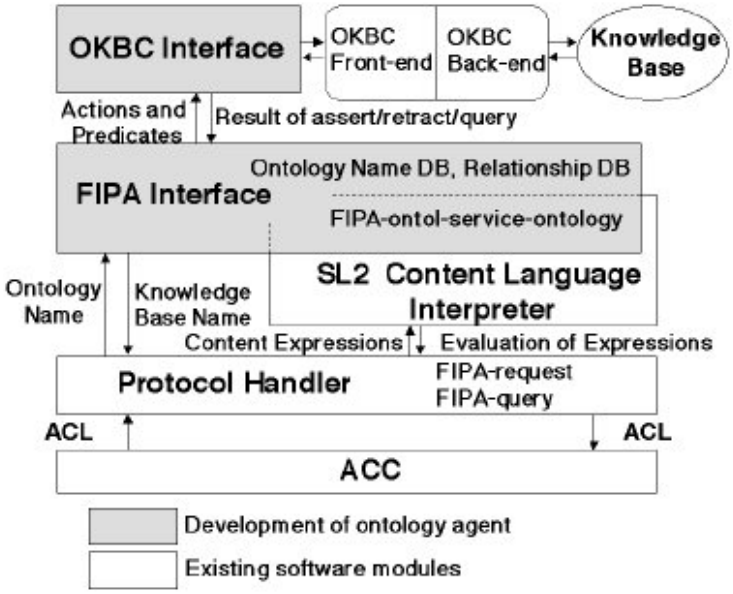


Figure 1: Structure of the platform regarding the ontology aspects, from [115]

The development of the OA is divided into two parts. The first part is an interface to the OKBC front-end. From the OKBC point of view, the OA is one of the front-end user applications. This part is also responsible for managing OKBC knowledge model and FIPA-meta-ontology. The second part is the FIPA interface where the agent wrapper is implemented. The FIPA interface includes functions such as ontology naming, ontology relationship and FIPA-ontol-service-ontology management. It utilizes existing interaction protocol handlers for FIPA-query and FIPA-request, which are prescribed in the specification for client agents that want to talk to the OA. The content language SL2 interpreter from the agent communication language libraries of the Comtec Agent Platform is also used to express the actions, objects and predicates in the ACL messages.

The COMTEC OA implements a subset of the services of a generic FIPA-compliant OA, in particular:

1. register an ontology in the framework;
2. operate over an ontology (create, delete frames, slot, modify the hierarchies);
3. answer queries about ontologies's structure, and their level of similarity.

The "translate" action of the FIPA-ontol-service-ontology is the only feature that is not implemented inside the specification. The OA registers with the DF expressing that it is unable to process the translate action from one ontology to another. This service is associated to the OKBC Interface.

The FIPA interface is an agent wrapper that takes care of generating and interpreting SL actions, predicates and ACL communicative acts based on appropriate interaction protocols. It also processes the registration with DF, and the management of ontology names and relationships between the ontologies.

This solution is among the best ones because it is really based on the FIPA standard. The problem is that the platform on which it was implemented is no longer available, and anyway it was based on an old version of the FIPA standard.

8.2 Implementation over the AgentService platform

Another implementation is that proposed by Vecchiola, Grosso, Boccalatte [125] who implemented a FIPA compliant framework called AgentService, based on the .NET platform. AgentService [2] integrates the functionalities of an Ontology Agent within the DF.

The AgentService design is however quite different from the FIPA standard. First of all, the OA is integrated in the DF, but it is only an implementative decision: actually AgentService integrates these two agents in a transparent way; only the DF knows that it is also the OA, but others agents anyway query it to find the OA, that in this case has the same address of the DF.

The services that have been implemented by the OA, are only a subset of the possible ones: the OA only implements the discovery and publication of the ontology and its maintenance, but neither the mapping or translation, nor the discovery of a good shared ontology.

The aim of this framework is mainly to help developers to create an ontology and then integrate it into the code of the agent, so it is more oriented to the "compile time" than to the "run time".

AgentService uses some tools (like Protégé [9] or Ms Visio©) to help the designer from the creation of the ontology to the development of an agent that can communicate over that ontology.

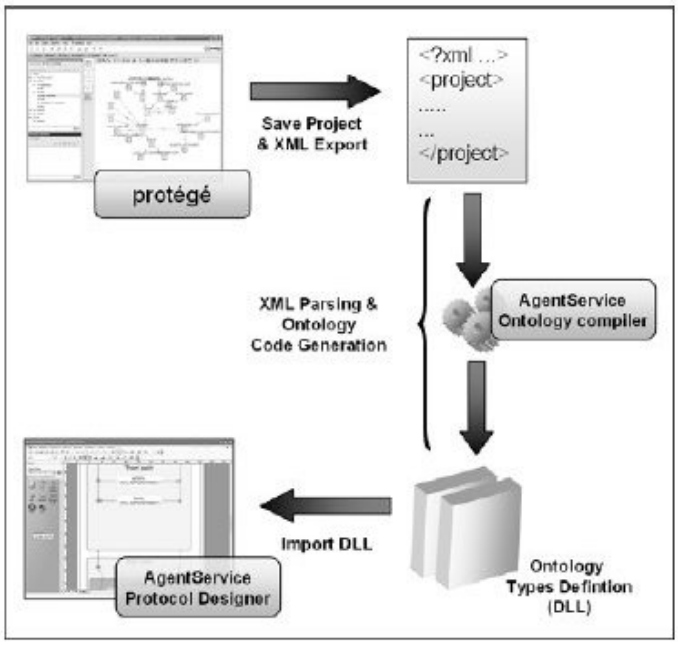


Figure 2: Generate ontology, from [125]

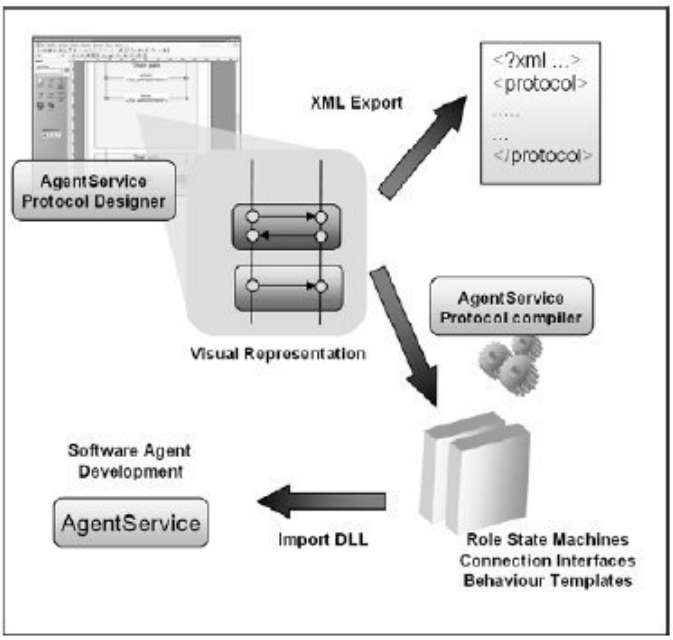


Figure 3: Generate interaction protocol, from [125]

To create an ontology Protégé is used: the ontology is then exported in an XML file and a tool integrated in AgentService automatically generates the corresponding object model and the source code, and compiles it into an assembly than can be used from the platform.

Then, using MS Visio©, is possible to model the interaction protocol (using AUML) based on the previously created ontology and again to export it in a format usable by AgentService and its agents. Figures 2 and 3 show this procedure.

The implementation of the OA provided by AgentService is among the few ones that follow the FIPA standard, but it offers more support to the design time integration of the ontology than to the run time integration: thus, it is useful but not very suitable for implementing open systems where heterogeneous/unknown agents interact and choose an ontology "on the fly".

8.3 Implementation over the JADE platform

JADE is of the most used FIPA-compliant platform, and offers some utilities to model the ontologies, but in the spirit of integrating the ontology into the agent's code. Thus, JADE supports the hard coding of the ontology both at design and compile time: all the entities of the ontology have to be transformed in classes and objects, so that a JADE agent may use them. The description of JADE support to integrate ontologies is descried in Section 8.4.4.

Anyway, this approach is not really useful when we are trying to create agents that can communicate with others automatically, after having agreed on an ontology. So JADE lacks a real support to the use of the ontology in open multiagent systems.

8.3.1 FIPA Like system

As far as the integration of a FIPA-compliant OA into JADE is concerned, there has been one attempt made by Obitko and Snáěl [98]. This implementation follows the standard in the sense that the design is straight based on the structure proposed by FIPA, but the authors decided to assume that the ontologies to be shared were in OWL-format: so they changed the meta-ontology in order to model the entities and the operations using OWL instead of OKBC.

The RDF, RDFS, and OWL languages are represented in the graph of triples (object, property, subject). The semantics of OWL is based on the description logic. Even when the modeling primitives look similar to OKBC, the semantics is different. Since Obitko and Snáěl intended to store OWL ontologies only, they had to adapt the language for describing actions performed by the ontology agent.

They proposed to use these ontologies:

1. OWL ontology itself: this plays a similar role as the FIPA-Meta-Ontology, and allows describing the ontologies and their parts. In fact, it defines all the ontological modeling primitives;

2. fipaMetaOnto ontology (based on OWL ontology): this defines one basic ontology for all FIPA-based agents, and serves as a base for the description of the FIPA multi-agent systems world. There is no direct counterpart of this ontology in FIPA proposals, the description is distributed in various FIPA documents;

3. fipaOntoService ontology: this extends the fipaMetaOnto ontology by identifiers needed for the ontology repository agent, and plays a similar role as the FIPA-Ontol-Service-Ontology ontology. This is an ontology that the FIPA ontology service agent should understand and that can be used for communication with this agent.

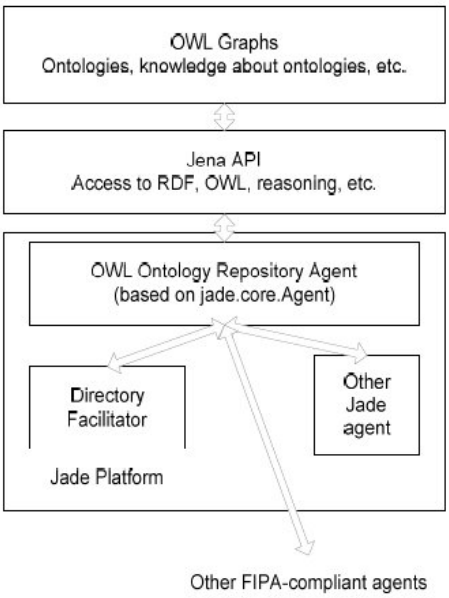


Figure 4: Structure of the system, from [98]

The class of actions can be defined as a special class in the fipaMetaOnto. For querying ontologies, RDQL [127] (RDF Data Query Language, a quite used W3C standard for extracting information from RDF graphs) can be used. The system is based on JADE and Jena [4] and implements the basic functionalities of the ontology services as specified in the FIPA proposal, i.e., the possibility to modify ontologies (assert and retract) and querying ontologies using RDQL.

Note that the OWL notation enables clear usage of multiple ontologies in one message in a standardized way, which is not possible with the FIPA standards. The architecture of the ontology repository implementation using JADE and Jena is illustrated in the Figure 4.

This implementation of the OA is well organized and closely follows the standard, except for the changes to the meta-ontology. This system may be completed by adding the primitives also for the OKBC standard, letting the OA being able to understand the queries for both ontologies.

8.3.2 Non FIPA compliant system

A good effort to design and implement a MAS with a support for ontology comes from Li Li, Baolin Wu and Yun Yang [80, 81]. Their work is mainly oriented towards the process of mapping and integrating ontologies: these functionalities are integrated into the MAS using a set of Agents which collaborate to offer them to the other agents. The agents which are involved in the process of ontology services are [1]:

1. User Agent (UA): assists the user in formulating his/her requests, posts queries (e.g. tasks) to the proposed system via the IA and visualizes the required results according to the user's requirement. The UA only knows the IA.

2. Interface Agent (IA): acts as an interface among agents in the MAS and the UA. Every agent knows about the IA.

3. Ontology Agent (OA): acts on behalf of the corresponding ontology, is in charge of ontology related tasks. It provides as much information of the ontology it acts on as possible. The OA operates over the ontology structure and the mapping result file. When a new ontology is loaded in the system, a corresponding OA is created to manage it.

4. Mapping Agent (MA): maps, if possible, concepts from an ontology to the concepts of a second ontology, using also the SA.

5. Similarity Agent (SA): maintains a thesaurus for the purpose of similarity. It holds a list of common words and synonyms of words

6. Query Agent (QA): operates over the mapping results to investigate ontology-understandable of heterogeneous ontologies after executing the ontology mapping.

7. Integration Agent (InA): merges two ontologies in a new one (in RDF format). It is based on the result of the ontology mapping.

[1] Although many of these agents have the same name as those in the FYPA system, they are completely different agents and offer completely different functionalities.

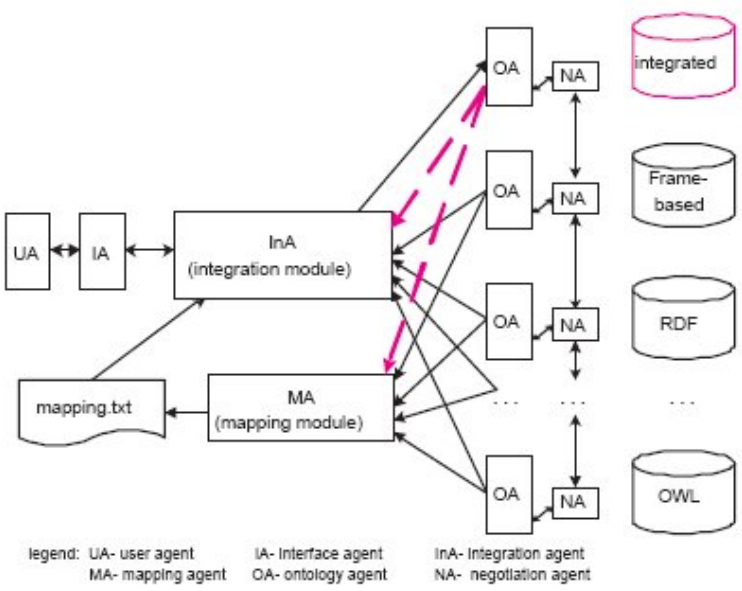


Figure 5: Organization of the Agents, from [80, 81]

8. Checking Agent (CA): checks the consistency of the integrated ontology (assuming all the given ontologies are consistent).

Figure 5 shows an overview of the system. This system is really interesting because it offers many services over the ontology, in particular those of mapping and integration. We do not enter into the details of the mapping and integration algorithm because it is out of our scope. From the point of view of the implementation of an ontology agent this is a good try, also if the structure is really far from the FIPA proposal because it is distributed among different agents, not centralized, and it seems to use only RDF/frame based ontologies (despite the figure shows an OWL ontology...).

8.4 Other integrations of ontologies in MASs

There are also other ways to include ontologies in a MAS, and they are left to the imagination of the designers and programmers. In this section we will present some original solutions to integrate ontologies and agents.

8.4.1 Transfer the complete ontology

The approach of Erdur and Dikenelli [48] to deal with dynamically changing and/or newly added global ontologies is to transfer these global ontologies at run time. By transferring ontologies at run time, agents can adapt themselves to the dynamically changing environments in such a way that:

1. agents managing the user interface can dynamically create the ontology dependent query interfaces;
2. agents managing information can dynamically create the ontology dependent information definition interfaces, which will assist in modeling the information in the repositories in an ontology dependent way;
3. agents can personalize the ontologies locally by extending them with additional rules.

Another good aspect is that this structure avoids the "bottleneck" of having only one agent managing all the ontologies, by making the overall system distributed.

The developed framework has been designed in an extensible style by collecting generic agent behaviors into a specific layer, called "Reusable Actions Layer". An ontology dependent behavior mentioned above is added to this layer together with other generic well-known behaviors, such as registering to directory facilitator (DF) etc. The developed framework has a layered architecture, which consists of the four layers shown in Figure 6.

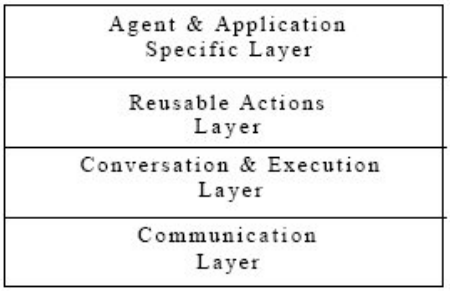


Figure 6: Structure of the system, from [48]

The "communication layer" and the "conversation and execution layer" provide the implementations necessary for having a FIPA compliant agent infrastructure.

The "reusable actions layer" contains the actions which are common to all agents, for example registering or deregistering with the Agent Management Service (AMS). It contains the actions that can be seen almost in all FIPA-compliant agent frameworks. However, this layer has been enriched with extra common actions, as those for initiating the transfer of ontologies, storing the

transferred ontologies into local knowledge bases, creating query interfaces dynamically from the transferred ontologies, inferencing on local ontologies: these are examples of reusable actions, which are related to the framework's ontology transfer capability.

The "Agent and Application Specific Layer" contains the actions which are specific to a certain application domain or to a specific agent type.

As ontology modeling language, the authors have also defined a model which includes the UML constructs necessary for ontology representation, OCL expressions and rules. They called this model as "XML based Ontology Transfer Model (XOT)" and defined an XML document type definition for it.

The content language is FIPA-rdf [53] (that is a FIPA specification that describes how the RDF language can be used as content language in a FIPA message) extended for expressing the basic elements of an ontology such as classes, attributes, relations, constraints.

All this structure has been made to avoid the usage of an unique agent and to let every agent create and manage the ontologies it needs: each agent can create its own instances of the ontologies it needs, without changing the global one. It may be a good solution but it loses the value of the Ontology agent, and it also uses a lot of non standard solutions (XOT, Fipa-rdf-extended) that make hard to reuse this system for other agents and platforms, and it forces the ontology to be in RDF format.

8.4.2 Ontology agent and Web Services

In [100] Pea, Sossa and Gutierrez tried to insert the ontology management in a MAS creating an agent, the Ontology Agent, encoded as a Web service, to expose its services also to the Internet. The OA carries out the management of the ontologies through an interface between the application Agents and the Ontologies. The Ontology Agent catches the events and messages that demand any components of the Ontologies, proceeds to interpret them, accomplishes the requested task and returns the answer. When an Agent receives a message, besides interpreting its content, requires knowledge of domain to fulfill its job. Thus, the Agents involved in the communication event have to share an Ontology of the domain. The MAS architecture is depicted in Figure 7.

The front-end layer is a Web Interface that offers the interaction tools to the user through: http://www.wolnm.org/mas/waInterface/wfInterface.aspx.cs. Nevertheless this page was no longer reachable at the time we wrote this thesis, but we cannot exclude that it has been updated and moved to another web address. The MAS encapsulates the middle-tier with all the functionalities by means of specialized Agents as: Interface, Encode, Decode, and Ontology. The back-end level corresponds to the Ontology repository integrated by the Ontologies and its respective Manager.

The Web Interface is built on the top of the MS Dot Net platform, composed by MS-Windows

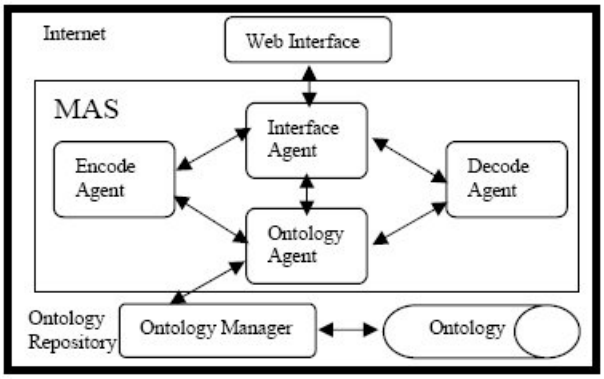


Figure 7: Structure of the MAS, from [100]

Server 2003, MS-Internet Information Server, Dot Net Framework 1.1, and MS-C#. The Interface Agent takes over of the interaction between the user and the MAS. So a communication flow is hold with the Web Interface in order to request services, send parameters, and receive responses.

The content of the messages submitted and interpreted among the Agents is encoded and decoded. The Encode Agent edits the parameters of the request and responses messages in XML format, and the Decode Agent navigates through the structure, interprets the elements and subelements, and accesses the values.

The Ontology Agent is the responsible to catch the requests, interpret them, forward the demands to the Ontology Manager in charge of the Ontology referenced and forward back the response from the Ontology manager.

The Ontology Repository is integrated by Ontologies and their respective Ontology Manager, which will insert, update, delete or consult the ontology elements. These ontologies are made available to the MAS only by the services provided by the OA.

The ontologies are in OWL format. Every Ontology Manager answers only to requests about the structure of the ontology or to orders to change the structure of the ontology, but no support is offered to the mapping, translating or more complex queries about two ontologies, so this agent seems to be more a coordinator of requests than a real Ontology Agent.

8.4.3 Ontology-services in an Electronic Institution

In [86] Malucelli and da Costa Oliveira inserted a support to the management of different ontologies in the field of electronic institutions (EI [49, 44, 69], briefly introduced in Section 2.1.2.6), using the platform ForEV [83] (that is a computing platform which includes and combines nego-

tiations methods in the context of Multi-Agent Systems, suitable for Virtual Enterprise formation scenario). In their work, they create an agent, called "ontology-services agent", which monitors all the conversations among the other agents in their EI (specifically developed to integrate and test the "ontology-services agent") and helps them in translating terms (concepts) from an ontology into another. In the scenario that the authors developed for testing the "ontology-services agent" capabilities, every agent in the EI has its own ontology: all of them have to be related to the same domain, but each agent has its own private ontology built in a private and unknown way. Also the ontology-services agent has its own ontology about the domain and it will use it to translate the terms. In the EI, four types of agents can act:

- Market agent: is the entity that facilitates the meeting of agents and supports the negotiation process;
- Builder agent: represents the enterprise interested in buying some components to build a final product;
- Provider agent: represents the enterprise interested in providing some kind of product;
- Ontology agent: monitors the whole conversation trying to help when it is necessary, providing some kind of services.

The builder agent sends a message to the market agent to inform it about which product components it wants to buy in order to build the corresponding product. Whenever a product is necessary, the market agent will send an announce to the provider agents. Several providers in the world may have these components with different prices and conditions. The Ontology agent will look at all the messages in the conversations, waiting for an "I don't know" from a Provider Agent: in this case the Provider is using a different ontology from the Builder, so they can't understand each other. The Ontology Agent will try to translate the terms in an equivalent-term of the ontology of the Provider, and than it will send the translation to the Provider. Therefore, in order to help to solve this incompatibility, the ontology agent:

- Searches for synonymous in its own ontology to find equivalent terms known by the receiving (Provider) agent;
- Searches for relations like "is", "is-a", "part-of", and "composed-by", in order to find out a different way of expressing the unknown term;
- Searches for characteristics of the component, which can help the provider ontology to find another component with the same characteristics;
- If all the previous steps are not enough, the ontology agent may not know the terms in appreciation, and then it looks in texts for new meanings and characteristics. A database of domain texts is used for the ontology agent to find terms.

Another service provided by the ontology-services agent is to help with different currencies. The ontology agent accesses a free web service that gives the currencies in real time. Dependencies can be another problem: some products already have dependencies, which are better to be known during the negotiation and the OA, always looking in its ontology, may help in this aspect.

The platform they use to implement the multi-agent system is JATLite [36] and the communication language is KQML. The ontology agent includes a basic domain ontology represented in XML.

This solution is not FIPA compliant because the OA is not directly interrogated by the other agents in the EI but it monitors all the conversations. This capability to "monitor and access" all the exchanged messages is the feature that makes this solution not completely generic (that is, not usable in any EI because the OA could not have the permission to "monitor" the messages in all the EIs). Moreover, the authors use the services already offered by the platform ForEv, that has a register agent responsible for building and maintaining the domain ontology. This agent is responsible for distributing the necessary information about this ontology when asked by some agent. So the described ontology-services is foreseen to be integrated in ForEv (that already offers some ontology management services), allowing the use of heterogeneous ontologies inside the platform.

8.4.4 Tools to hard-code the ontologies: the JADE support

JADE offers a support to integrate ontologies [121] into the code of an agent, that is, to represent the entities of the ontologies in a format compliant with the framework. So it tries to transform the ontology in a set of classes that can be used by the JADE agent. The type of support is studied for the frame-based ontologies.

In order for JADE to perform the proper semantic checks on a given content expression of a message, it is necessary to classify all possible elements in the domain of discourse (i.e. elements that can appear within a valid sentence sent by an agent as the content of an ACL message) according to their generic semantic characteristics. This classification is derived from the ACL language defined in FIPA that requires the content of each ACLMessage to have a proper semantics according to the performative of the ACLMessage. More in details, at the first level it distinguishes between predicates and terms.

- Predicates are expressions that say something about the status of the world and can be true or false e.g. (Works-for (Person :name John) (Company :name TILAB)) stating that "the person John works for the company TILAB". Predicates can be meaningfully used for instance as the content of an INFORM or QUERY-IF message, while would make no sense if used as the content of a REQUEST message.

- Terms are expressions identifying entities (abstract or concrete) that "exist" in the world and that agents talk and reason about. They are further classified into:
  - concepts, i.e., expressions that indicate entities with a complex structure that can be defined in terms of slots, e.g. (Person :name John :age 33). Concepts typically make no sense if used directly as the content of an ACL message. In general they are referenced inside predicates and other concepts such as in (Book :title "The Lord of the rings" :author (Person :name "J.R.R. Tolkjien"));
  - agent actions, i.e., special concepts that indicate actions that can be performed by some agents e.g., (Sell (Book :title "The Lord of the rings") (Person :name John)). It is useful to treat agent actions separately since, unlike "normal" concepts, they are meaningful contents of certain types of ACLMessage such as REQUEST. Communicative acts (i.e. ACL messages) are themselves agent actions;
  - primitives, i.e., expressions that indicate atomic entities such as strings and integers;
  - aggregates, i.e., expressions indicating entities that are groups of other entities, e.g. (sequence (Person :name John) (Person :name Bill));
  - Identifying Referential Expressions (IRE), i.e. expressions that identify the entity (or entities) for which a given predicate is true, e.g. (all ?x (Works-for ?x (Company :name TILAB)), identifying all the elements x for which the predicate (Works-for x (Company :name TILAB)) is true, i.e. all the persons that work for the company TILAB). These expressions are typically used in queries (e.g. as the content of a QUERY-REF message) and require variables;
  - variables, i.e., expressions (typically used in queries) that indicate a generic element not known a priori.

Exploiting the JADE content language and ontology support to make agents talk and reason about "things and facts" related to a given domain goes through the following steps.

1. Defining an ontology including the schemata for the types of predicate, agent action and concept that are pertinent to the addressed domain.
2. Developing proper Java classes for all types of predicate, agent action and concept in the ontology.
3. Selecting a suitable content language among those directly supported by JADE. JADE can be easily extended to support new user-defined content languages.
4. Registering the defined ontology and the selected content language to the agent.

5. Creating and handling content expression as Java objects that are instances of the classes developed in step 2 and let JADE translate these Java objects to/from strings or sequences of bytes that fit the content slot of ACLMessages.

Using Java objects to represent a content expression is very convenient for managing the information included in that content expression, but there are, however, some cases where this can create problems. For example, if the ontology has many elements, we will have to create and deal with the same number of classes (imagine if they are 1000...). If we want to model multiple inheritance between concepts of the ontologies, we have to use an interface instead of classes (because Java does not support multiple inheritance). Furthermore, in order to create queries it is necessary to specify variables such as in QUERY-REF (All ?x (Owns (agent-identifier :name Seller) ?x), but such a content expression can't be translated into a Java object.

For these reasons JADE provides another (less convenient, but more general) way of representing content expressions: each element can be represented as an abstract descriptor that includes:

- a type-name indicating the actual type of the element;
- a number of named slots holding the attributes of the element.

This is to say that, e.g., the concept (Person :name Giovanni :age 33) can be also represented as an instance of the AbsConcept class (an abstract descriptor representing a concept) where the type-name is set to "Person", the slot named "name" is set to "Giovanni" and the slot named "age" is set to 33.

All this work may be made by hand or can be made automatically by tools, as BeanGenerator.

BeanGenerator [3] is a plug-in for Protégé, and helps the programmer to generate code representing the ontology that (s)he wants to use. With this program, the developer can generate Java files representing an ontology that can be used by the JADE environment and agents. Within Protégé the developer can import and export RDF and RDFS. With the BeanGenerator tool, (s)he can generate FIPA/JADE compliant ontologies from RDF(S), XML and Protégé projects.

This tool supports the programmer in the generation of the classes needed by JADE to integrate an ontology in the agent code. Its limitation is that it deals only with the RDF-ontologies.

8.5 Our Ontology Agent

In this section we describe our implementation of an OA for JADE, able to provide both "standard" (those foreseen by FIPA) and new ontology matching services, as well as services for comparing one or more alignments to a reference one. In particular, our OA offers services for

ontology matching via upper ontologies, a new approach that has proven to give good results in precision and recall [88].

Since most of the existing FIPA-compliant OAs provide services for ontology discovery and interrogation, whereas none of them provides a service for ontology matching, we started our research by implementing the latter, in order to complement existing proposals. The missing services have been added to our OA by Federico Mulattieri in his bachelor Thesis [93]. As other researchers, we deviate from the FIPA specification since we assume that ontologies are represented in OWL instead than OKBC.

This research activity has been carried out as a joint work with Locoro, Mascardi, Rosso and Mulattieri. The work has been organized as shown later:

- the author of this Ph.D. Thesis focused above all on the design of the Ontology Agent
- Locoro, Mascardi, Rosso designed and developed the matching algorithm, as described in [88]
- Locoro, Mascardi developed the OA and the service that uses their algorithm for the ontology matching
- Mulattieri, in his bachelor thesis, added the missing standard services to the OA developed so far.

In the next sections we describe the developed OA and the ontology matching service integrated in it.

8.5.1 Ontology Matching

A formalization of the ontology matching process can be found in [50]. Quoting the authors, a matching process can be defined as *"a function f which takes two ontologies o and o', an input alignment a, a set of parameters p and a set of oracles and resources r, and returns an alignment a' between o and o'"*.

A correspondence (also named "mapping") between an entity e belonging to ontology o and an entity e' belonging to ontology o' is a 5-tuple $< id, e, e', R, conf >$ where:

- id is a unique identifier of the correspondence;
- e and e' are the entities (e.g. properties, classes, individuals) of o and o' respectively;
- R is a relation such as "equivalence", "subsumption", "disjointness", "overlapping", holding between the entities e and e';

- $conf$ is a confidence measure (typically in the [0;1] range) holding for the correspondence between the entities e and e'.

An alignment of ontologies o and o' is a set of correspondences between entities of o and o'.

As indicators for measuring how good an alignment is, precision, recall and F-measure adapted for ontology alignment evaluation [45] are used in this research.

Precision is defined as the number of correctly found correspondences with respect to a reference alignment (true positives) divided by the total number of found correspondences (true positives and false positives) and recall is defined as the number of correctly found correspondences (true positives) divided by the total number of expected correspondences (true positives and false negatives). A perfect precision score of 1.0 means that every correspondence computed by the algorithm was correct (correctness), whereas a perfect recall score of 1.0 means that all correct correspondences were found (completeness).

To compute precision and recall, the alignment a returned by the algorithm is compared to a reference alignment r.

Precision is given by the formula

$$P(a,r) = \frac{\mid r \cap a \mid}{\mid a \mid}$$

whereas recall is defined as

$$R(a,r) = \frac{\mid r \cap a \mid}{\mid r \mid}$$

We also use the harmonic mean of precision and recall, namely F-measure:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Among the matching techniques, in the next only those that fall under the "Granularity / Input Interpretation" classification described in [50], based on the granularity of the matcher and on the interpretation of the input information, are considered.

- *String-based methods.* These methods measure the similarity of two entities just looking at the strings (seen as mere sequences of characters) that label them. Among them we may cite substring distance, where two strings are compared to find the longest common substring, n-gram distance [24], where two strings are the more similar the more n-grams (sequence of n characters) they have in common, and SMOA measure [113], which is a function of the commonalities (in terms of substrings) as well as of differences between two strings.

- *Language-based methods.* These methods exploit natural language processing techniques to find the similarity between two strings seen as meaningful pieces of text rather than sequences of characters. Some of them exploit external resources like WordNet, and exploit the semantic relations that it offers to compute the correspondences.

8.5.2 Upper Ontologies and their Application to Ontology Matching

An upper ontology (also named top-level ontology, or foundation ontology) is "*an attempt to create an ontology which describes very general concepts that are the same across all domains*" [134]. Few upper ontologies exist: BFO [61], Cyc [78], DOLCE [57], GFO [64], PROTON [35], Sowa's ontology [112], and SUMO [95]. They vary in dimension, ranging from the 30 classes of Sowa's ontology to the 300,000 of Cyc, in representation language (OWL, KIF [17], First Order Logic), in structure (monolithic vs. decomposed into modules), and in developed applications. Nevertheless, all of them describe general concepts (also named "classes") and share the aim to have a large number of ontologies accessible under them.

In their previous work, Mascardi, Locoro and Rosso implemented different algorithms that used upper ontologies for boosting the ontology matching process [88]. They run experiments with SUMO-OWL (a restricted version of SUMO translated into OWL), OpenCyc (the open version of Cyc, which is a commercial ontology), and DOLCE. The experiments demonstrate that when the "structural matching method via upper ontology" uses an upper ontology large enough (OpenCyc, SUMO-OWL), the recall is significantly improved and the precision is kept w.r.t. not using upper ontologies. Instead, the "non structural matching method" via OpenCyc and SUMO-OWL improves the precision and keeps the recall. The "mixed method", that combines the results of structural alignment without using upper ontologies and structural alignment via upper ontologies, improves the recall and keeps (improves, with OpenCyc) the F-measure, whatever the upper ontology used.

8.5.3 Services offered

The OA we have implemented in JADE provides the following services:

1. matching two OWL ontologies through a direct matching;
2. matching two OWL ontologies via an upper ontology (represented in OWL too);
3. evaluating an alignment against a reference alignment.

plus the standard ones:

1. registering an ontology;
2. downloading an ontology;
3. searching an ontology using tags;
4. executing query on an ontology;
5. modifying an ontology;
6. matching concepts between ontologies.

The standard services that manage the ontologies, as well the queries over them, have been developed using the Jena libraries [4]. Jena is a Java open source framework that supports the development of applications for the Semantic Web, and that lets the developer use RDF, RDFS, OWL and SPARQL.

In this section, the algorithms that realize the matching services of the OA are described. It is worth specifying that in this matching algorithms the authors (Locoro, Mascardi, Rosso) only consider concepts as entities to match, and equivalence as relation.

Many algorithms for ontology matching exist, and some of them have been implemented and are available to the research community: for example, the Alignment API developed by J. Euzenat and his team at INRIA-Rhône Alpes is available at `http://alignapi.gforge.inria.fr/` under GNU Lesser General Public License. It provides string-based and simple language-based ontology matching methods, as well as methods for computing precision and recall of a given alignment w.r.t. a reference one. Surprisingly, none of them has been integrated into an OA. Thus, to the best of our knowledge, no existing OAs offer services for ontology matching.

In order to be compliant with the above mentioned Align API, and to be able to exploit some of the methods it provides, the alignments are represented in RDF and their format is like the one shown below.

```
<rdf:RDF .. namespaces here ...>
<Alignment>
<xml>yes</xml>
<level>0</level>
<type>**</type>
<time>15</time>
<method>StringDistAlignment</method>
<onto1>file:///ka.owl</onto1>
<onto2>file:///edumit.owl</onto2>
<uri1>file:///ka.owl</uri1>
<uri2>file:///edumit.owl</uri2>
  ...
<map> <Cell>
<entity1 rdf:res=ka.owl#Book/>
<entity2 rdf:res=edumit.owl#Book/>
<relation>=</relation>
```

```
<measure>1.0</measure>
</Cell> </map>
<map> <Cell>
<entity1 rdf:res=ka.owl#TechnicalReport/>
<entity2 rdf:res=edumit.owl#Techreport/>
<relation>=</relation>
<measure>1.0</measure>
</Cell> </map>
  ...
</Alignment>
</rdf:RDF>
```

With respect to the definition provided in Section 8.5.1, the found correspondences are simpler due to the lack of the correspondence identifier, which is strictly necessary only in case we need to uniquely identify them (e.g. when storing them in a repository).

8.5.4 Direct Matching

The direct matching service is implemented by a function that the is called

$parallel\_match(o, o', \{WordNet\}, th)$

because it runs in parallel different "standard" matching methods, and combines their outputs. The matching methods that are used are the *substring*, *n-gram*, *SMOA* and *WordNet* ones that are introduced in Section 8.5.1. The implementation of these methods offered by Euzenat's Align API has been used,in particular the Alignment API version 3.1 one. The used methods are *StringDistAlignment* (that provides `subStringDistance`, `ngramDistance` and `smoaDistance` string metrics), and *JWNLAlignment* (that computes a substring distance between two concepts exploiting their WordNet 3.0 synsets). The th parameter is used as a threshold for cutting all correspondences below it. In the experiments, they were set at 0.5. To obtain a final alignment from the ones obtained by the individual matching methods, an $aggregate$ function that composes the four alignments by making the union of all their correspondences is used. In case different alignments produced correspondences between the same concepts c and c', $aggregate$ keeps the correspondence with the highest confidence measure.

8.5.5 Matching via Upper Ontology

For matching two ontologies o and o' via an upper ontology uo the authors compute the

$parallel\_match(o, uo, \{WordNet\}, th)$ and $parallel\_match(o', uo, \{WordNet\}, th)$

obtaining two alignments between o and uo, and o' and uo respectively. These two alignments are given in input to a $merge(a, a')$ function. $Merge$ produces the final alignment between o and o' by combining the correspondences of o-uo and o'-uo in such a way that: if $\exists$ a correspondence

$< c, c_u, r, conf1 >$ in o-uo and $\exists$ a correspondence $< c', c_u, r, conf2 >$ in o'-uo, then the $merge$ function creates a new correspondence $< c, c', r, conf1 * conf2 >$ and adds it to the final alignment.

8.5.6 Alignment Evaluation

The Alignment API provided by Euzenat and colleagues offers a *PrecEval* method for measuring the goodness of an alignment based on precision, recall and F-measure.

The *PrecEval* method has been integrated into our OA; an example of evaluation produced by the OA is given by the following RDF fragment:

```
<rdf:RDF ..namespaces here ..>
  <map:output rdf:about=''>
    <map:in1 rdf:resource="ka.owl"/>
    <map:in2 rdf:resource="edumit.owl"/>
    <map:precision>0.097</map:precision>
    <map:recall>0.084</map:recall>
    <map:fMeasure>0.09</map:fMeasure>
    <map:nbcorrect>7</map:nbcorrect>
   <map:nbfound>72</map:nbfound>
  </map:output>
</rdf:RDF>
```

In order to test the behavior of our OA a Request Agent (RA) has been implemented: it acts as an interface between the user and the OA, and allows the user to request services to the OA. The RA receives a set of parameters from the user, and saves them for later use. These parameters are:

- the URI of the OWL file containing the first ontology to match, o;
- the URI of the OWL file containing the second ontology to match, o';
- the URI of the file where the computed alignment will be stored;
- the URI of the file containing the reference alignment for performing an evaluation of the computed alignment;
- the matching method ("direct matching" or "matching via upper ontology");
- a further optional parameter providing the URI of the file containing the upper ontology in OWL. This parameter is needed only in case of matching via upper ontology.

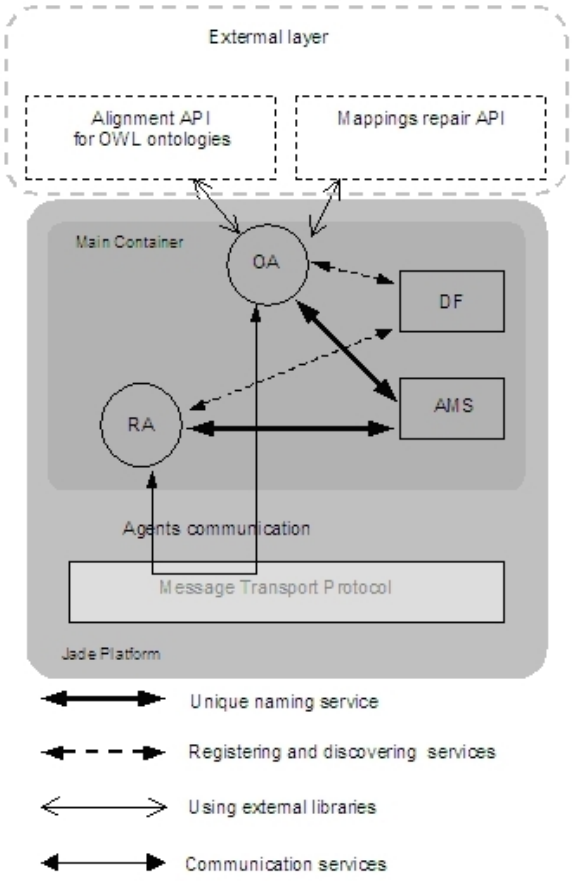


Figure 8: Architecture of the Ontology Agent integrated into JADE.

The system architecture is depicted in Figure 8.

During its start up phase, RA searches and identifies an OA inside the JADE Platform.

After having received the parameters from the human user, and having found OA, RA:

- sends as many INFORM messages to the OA, as the parameters it wants to send to OA; the parameters are sent as the message content (one parameter for each message); the conversation id of the message is used to keep track of the conversation just started;
- sends a REQUEST message asking for the matching and evaluation services (which, in this first prototype, are always coupled); the ontologies to match, and the method to use, are those passed in the previous INFORM messages.

OA starts its life with the registration of its ontology services to the DF. When RA begins the

interaction phase, OA reacts as follows:

- it receives all the parameters from the RA and sends an ACL message in reply to each message, setting the same conversation id for correctly keeping track of the conversation, after having successfully received and saved each parameter;
- on reception of a REQUEST message, it runs the required matching method on the ontologies specified in the previous interaction steps with the RA;
- sends an INFORM message to notify the RA of the accomplishment of the matching service and to communicate the alignment file URI and the evaluation file URI; in case something goes wrong, it sends a FAILURE message to the RA.

Both agents terminate after the completion of these activities.

OA and RA have been structured according to JADE recommendations for the development of agents. RA's `setup()` method includes the steps to save parameters from the command line, the instantiation of a `wakerBehavior` object in order to find the OA via DF query and the call to the `RequestPerformer` behavior. This one includes the `action()` method where the agent sends the parameters, sends the alignment and evaluation requests, and waits for the results. It has been implemented as a generic `Behavior` class.

OA's `setup()` method starts with the registration of OA's services to the DF and the instantiation of its two behaviors which are of type `ReceiveParameters` and `Align` respectively. The `action()` method of `ReceiveParameters` is dedicated to receive the parameters and has been implemented as a `OneShotBehavior` class. The `action()` method of `Align` first waits for an ontology matching request. After the request message has been received, the matching and evaluation activities are performed, and the result is notified to the sender. This class has been modeled extending the generic `Behavior` class.

To better synchronize the exchange of messages, each of them is received by the two agents in blocking mode and the `MessageTemplate` class has also been used to deal with conversation id patterns.

JADE 3.6 has been used for the development of the MAS, consisting of the OA and of the RA.

In order to verify the feasibility of this approach, a large set of experiments has been executed. Only two of them are discussed in details; the other ones have been carried out in a similar way.

In the first experiment, the RA asks the OA to execute a direct alignment between

Ka (`protege.cim3.net/file/pub/ontologies/ka/ka.owl`) and

Bibtex (`oaei.ontologymatching.org/2004/Contest/304/onto.rdf`).

Ka has 96 concepts dealing with the academic domain, including Event (Activity, Meeting, Conference, Workshop), Publication (concepts similar to those of the *Bibtex* ontology), Organisation (Department, Enterprise, Institute, University), ResearchTopic, whereas *Bibtex* has only 15 concepts including Article, Book, Conference, Manual, Person, Publisher, etc.

Before using *Ka* and *Bibtex* in the experiments, they have been pre-processed by hand in order to remove properties and individuals that are not used in the matching algorithms. A reference alignment between *Ka* and *Bibtex* has been created by hand because the authors wanted to evaluate how good the alignments resulting from the automatic matching methods were, w.r.t. the reference one.

The URIs of the two ontologies to match are passed as input parameters to RA together with the URI of the file that had to contain the alignment, the URI of the file that contains the reference alignment, and the chosen method (direct alignment).

The second experiment also aimed at aligning *Ka* and *Bibtex*, but using the "matching via upper ontology" method. The used upper ontology is SUMO-OWL, a lossy translation into OWL of the SUMO ontology, whose full version is encoded in KIF.

Method	**Found**	**Correct**	**Prec.**	**Recall**	**F-meas.**
Direct	27	7	0.26	**0.08**	**0.13**
SUMO-OWL	9	6	**0.67**	0.07	**0.13**

Table 1: Matching Ka and Bibtex: results summary.

The results of these two experiments are shown in Table 1, that summarizes the total correspondences found w.r.t. the correct ones and the precision, recall and f-measure computed by the *PrecEval* method integrated into our OA.

While running this experiment, the JADE sniffer agent has been used to follow the interactions between RA and OA. A screen shot of the "sniffed" messages is given in Figure 9. The messages exchanged in the first and second experiment are very similar. In both experiments OA and RA exchange messages to pass and save parameters. When the parameter negotiation phase ends, the RA sends a request to OA for performing the matching and the evaluation. The task has been successfully executed in both experiments. As a consequence OA sends an INFORM message to RA to inform it of the successful outcome of the matching, and both agents terminate.

The first experiment uses a direct matching method that took less than a minute to complete, while the second one required about 10 minutes due to the usage of a more sophisticated matching algorithm involving a large upper ontology (SUMO-OWL has 4,393 concepts). The experiments have been executed on a HP Pavillon Notebook with Intel Core Duo T2250 processor, 1.73 GHz of clock, 2 GB of RAM, and Windows XP. From a comparison of the alignments computed by the OA in the two experiments, which are reported in Table 1, it turns out that matching *Ka*

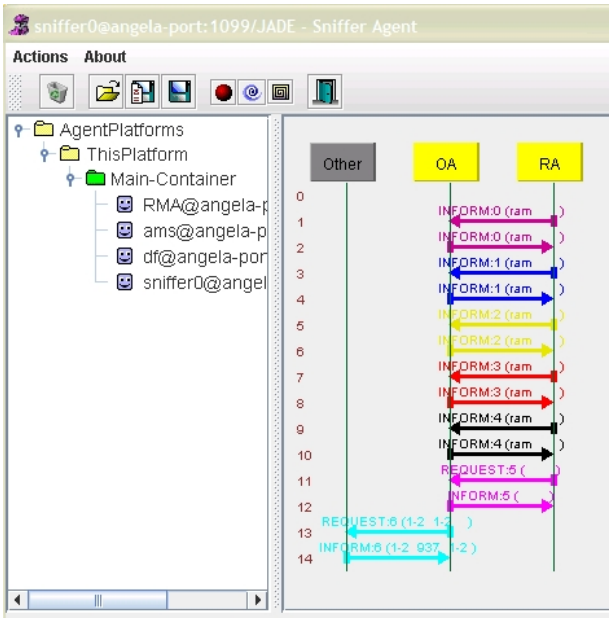


Figure 9: A sniffer screen shot with OA and RA agents.

and *Bibtex* via SUMO-OWL gives the best precision (67% against 26% of the direct alignment), while recall and F-measure are the same (recall: 7% against 8%; F-measure: 13% for both).

Other 9 more tests have been executed. Besides *Ka* and *Bibtex*, the ontologies that have been used in the experiments and the URLs from where they may be downloaded are:

- *Agent*, `212.119.9.180/Ontologies/0.2/agent.owl`
- *Biosphere*, `sweet.jpl.nasa.gov/ontology/biosphere.owl`
- *Ecology*, `wow.sfsu.edu/ontology/rich/EcologicalConcepts.owl`
- *Food*, `silla.dongguk.ac.kr/jena-owl1/food`
- *Geofile*, `www.daml.org/2001/02/geofile/geofile-ont.daml`
- *HL7_RBAC*, `lsdis.cs.uga.edu/projects/meteor-s/wsdl-s/ontologies/HL7_RBAC.owl`
- *MPEG7*, `dmag.upf.es/ontologies/2003/03/MPEG7Genres.rdfs`
- *Restaurant*, `guru-games.org/ontologies/restaurant.owl`

- *Resume*, statistic.gunadarma.ac.id/research/WorkGroupInformationSystem/DownLoad/onto_colection/resume.owl
- *Space*, 212.119.9.180/Ontologies/0.3/space.owl
- *Subject*, www.library.yale.edu/ontologies/subject.owl
- *Top-bio*, www.co-ode.org/ontologies/basic-bio/top-bio.owl
- *Travel*, lsdis.cs.uga.edu/projects/meteor-s/downloads/Lumina/ontology/travelontology.owl
- *Vacation*, www.guru-games.org/ontologies/vacation.owl
- *Vertebrate*, www.co-ode.org/ontologies/basic-bio/basic-vertebrate-gross-anatomy.owl (Vertebrate has been reduced by hand when running our experiments)

The experiments run consisted of matching the following couples of ontologies:

- **1:** Ka - Bibtex;
- **2:** Biosphere - Top-bio;
- **3:** Space - Geofile;
- **4:** Restaurant - Food;
- **5:** MPEG7 - Subject;
- **6:** Travel - Vacation;
- **7:** Resume - Agent;
- **8:** Resume - HL7_RBAC;
- **9:** Ecology - Top-bio;
- **10:** Vertebrate - Top-bio

The obtained results are summarized in Table 2. In 7 experiments out of 10, matching via SUMO-OWL allowed to obtain the best precision. In many cases, the recall of the two methods can be compared even if, in some experiments, the direct matching performs definitely better. The suitability of the "matching via upper ontologies" method strictly depends on the ontologies to match. From these experiments we have learnt that, for example, the type of English words used by both upper and matched ontologies counts, as well as the terminology used by both upper and matched ontologies.

Test	Method	Found	Correct	Prec.	Rec.	F-meas.
1	Manual	83	83	1.00	1.00	1.00
1	Direct	27	7	0.26	**0.08**	**0.13**
1	SUMO-OWL	9	6	**0.67**	0.07	**0.13**
2	Manual	604	604	1.00	1.00	1.00
2	Direct	26	6	**0.23**	**0.01**	**0.02**
2	SUMO-OWL	2	0	0.00	0.00	0.00
3	Manual	513	513	1.00	1.00	1.00
3	Direct	164	38	0.23	**0.07**	**0.11**
3	SUMO-OWL	49	22	**0.45**	0.04	0.08
4	Manual	1041	1041	1.00	1.00	1.00
4	Direct	107	12	0.11	0.01	0.02
4	SUMO-OWL	82	28	**0.34**	**0.03**	**0.05**
5	Manual	637	637	1.00	1.00	1.00
5	Direct	323	94	0.29	**0.15**	0.20
5	SUMO-OWL	224	93	**0.42**	**0.15**	**0.22**
6	Manual	262	262	1.00	1.00	1.00
6	Direct	50	19	**0.38**	**0.07**	**0.12**
6	SUMO-OWL	26	8	0.31	0.03	0.06
7	Manual	1122	1122	1.00	1.00	1.00
7	Direct	157	58	0.37	**0.05**	**0.09**
7	SUMO-OWL	71	31	**0.44**	0.03	0.05
8	Manual	295	295	1.00	1.00	1.00
8	Direct	127	36	0.28	**0.12**	0.17
8	SUMO-OWL	60	34	**0.57**	**0.12**	**0.19**
9	Manual	308	308	1.00	1.00	1.00
9	Direct	88	28	**0.32**	**0.09**	**0.14**
9	SUMO-OWL	140	17	0.12	0.06	0.08
10	Manual	19	19	1.00	1.00	1.00
10	Direct	12	2	0.17	**0.11**	0.13
10	SUMO-OWL	7	2	**0.29**	**0.11**	**0.15**

Table 2: Complete results of our experiments

8.6 Evaluation and future work on the Ontology Agent

Our OA is able to produce alignments between two OWL ontologies via direct matching or via an OWL version of an upper ontology. It evaluates the obtained alignment with respect to a given reference one. The exploitation of an upper ontology for performing an alignment between two ontologies is a new approach that Mascardi, Locoro and Rossi described in [88]. This approach represents an original contribution to the ontology matching process. For what concerns their decision to match concepts only, and to limit themselves to considering the equivalence relation, the intention is to extend the matching methods towards the exploitation of properties and individuals, and to take into consideration at least subsumption and disjunction relations.

Our OA can be improved through a deeper analysis of the FIPA-Agent-Management ontology in order to see if it is feasible to join the description of the services provided by OA with concepts of the JADE `BasicOntology`, thus allowing the matching process, the evaluation and the correspondences repair services to be a part of the JADE Agents semantic.

Future extensions we would like to carry on are:

- the addition of the translation service among different languages;
- the addition of the "similarity" among ontologies, to specify if two ontologies are, for example, equivalent or if one is an extension of th other and so on;
- the modeling of more complex behaviors which would better reflect the alignment, evaluation and correspondences repair tasks (i.e. parallel behaviors for string based and language based partial alignments which could be nested inside a sequential behavior able to produce the final alignment and the merged alignment via an upper ontology);
- the extension of OA life to prolong the availability of these services for the whole time the platform is running;
- the exploitation of FIPA standard interaction protocols to manage the conversation between our agents;
- the separation of the alignment process from the evaluation one by providing different requests for alignment, evaluation and correspondences repair (exploiting a modular architecture).

8.7 Integration of Ontology Agent in the protocols developed: an analysis

Since the FYPA and the MAD MASs have been developed in JADE, the integration of our OA in them could easily be achieved. Nevertheless at this stage of the research our OA was needed in none of them, because of their very specific features that allowed us to design the entire MASs by ourselves, hence avoiding problems related to MAS openness and heterogeneity of agents.

Anyway, we could suppose to make our protocols more generic and so to need in that case the services of an Ontology Agent.

For example let us suppose that our "Stand-alone FYPA" lets Foreign Trains enter the station or, in other words, lets Agents developed by a third subject to enter the negotiation protocol. This scenario could for example appear in Border Stations (those on the nation frontier) where Trains from different nations can arrive. In this case we could use an ontology that allows different Trains understand each others, learning how to participate to the protocol, which terms must be used, which configuration time-out must be applied and so on. All this architecture should be obviously integrated in a structure like an Electronic Institution.

This scenario is also very similar to the one where trains come from a station not managed with FYPA: in this situation too, the incoming trains could be helped by the Ontology Agent to understand how the FYPA protocol is organized and how to join the negotiation, avoiding a different management of those trains coming from not FYPA systems.

The sharing of a common ontology or the presence of a way to translate concepts among different ontologies could clearly be a pre-requisite of the system, and in that case we could think integrating our Ontology Agent in the FYPA system.

In a similar way we could think to insert ontologies also in the MAD system: in this case we should provide an open system where many software applications produced by different developers (Companies) cooperate to achieve a final goal (that is, to make the overall system run). In this situation probably to analyze the log files (assumed that we have the permissions) could be difficult because every system will use different terms to describe an anomaly (using a proprietary ontology): we could insert the Ontology Agent in the system and increase the agents capabilities with the ability to use ontologies to analyze the log files and to monitor data coming from different sources. The Ontology Agent might help agents to make an alignment among ontologies to gain a real overall monitoring of the system's global state.

Moreover, ontologies could be exploited to configure the MAD MAS in a network-independent way. Let us suppose to create an ontology that describes a generic network, that is, the organization of PCs and of processes (how PCs are connected and which processes run on which machines). In this way, a network can be represented by an instance of an ontology. Then, we could created a new agent that is able to read from this ontology and, at run time, it could create

all the agents (PMA, CMA etc.) needed to manage this specific network.

All the agents in the MAD MAS could be modified to read from the ontology to understand, for example, which processes they have to manage and how they can do this, that is, also the Prolog rules could be integrated in the ontology.

In this way the MAD MAS could manage every network, because it only needs the ontology representing it. This could be a really good improvement of the MAD MAS, that can become independent from the network and also from the considered processes.

Chapter 9

Conclusions and considerations

Let us report here the main goal of this thesis, as already described in the first chapter:

> *The primary goal of the thesis is the development of a new interaction protocol for solving a concrete resource allocation problem that shows interesting features from both an industrial and an academic point of view. To this aim, we generalize a real problem proposed by Ansaldo STS, we provide a sound mathematical model for the generalized problem, we design a MAS where agents negotiate in order to solve the generalized problem, and finally we implement and test the designed MAS. Providing an insight of how ontologies can be exploited in systems consisting of negotiating agents is the secondary goal of the thesis.*

In the next two sections we briefly recall the results we achieved while working for reaching our goal, and the publications related to it. Our plans for the future developments of this thesis are also discussed.

9.1 Achieved results

The main goal of this thesis was to design a new negotiation protocol to manage the real time allocation of a set of resources (railway tracks) to a set of agents (trains in a station). We designed the protocol FYPA that coordinates a set of agents to find dynamically an allocation for a set of interconnected resources. The protocol was devised starting from an industrial case study, provided by Ansaldo STS, but it can be generalized as a resource allocation problem solved with a negotiation among agents.

We developed a system that is able to interface with Ansaldo STS application and another system that is completely independent from the Ansaldo STS one.

Furthermore we analyzed the state of the art of the use of negotiation to solve MARA problems and we made a comparison with some protocols similar to ours.

The results of the work on FYPA have been described in [32, 30, 28, 29].

In the field of negotiation in MASs, we also designed and developed a MAS that is able to monitor a set of processes running on different machines connected by a network. In this system we exploited the hierarchical organization of the MAS.

The results of the "MAD MAS" project are described in [87, 31].

As far as the integration of ontologies in MASs is concerned, we stated our research carrying on a detailed analysis of the state of the art: we analyzed existing proposals to exploit ontologies in MASs, both following the FIPA standard and using different architectures and technologies. Then we designed a specific Ontology Agent, as close as possible to the FIPA proposal, that afterwards has been developed by the other members of our research group.

The results of this research activity are described in [27, 25].

9.2 Future work

Now that the systems and protocols described in this thesis have reached a stable version, we can start to think over improving and extending them.

In this section we illustrate some activities we would like to carry out, besides from those already discussed in Sections 3.5, 5.6, 8.6, 8.7.

The "MAD MAS" project, that Ansaldo STS used as a testbed to evaluate the multiagent technology but is not currently exploiting for its real scenarios, could represent a good starting point for carrying on research on similar problems. When we designed the "MAD MAS", in fact, we found applications both in the academic area and in the industrial one that were very close to "MAD MAS". It would be interesting to look for other similar case studies in order to have a more comprehensive understanding of the state-of-the-art, and, based on that, designing an "improved MAD MAS" exploiting some complex organizations among those described in Section 2.1.2.

AS far as FYPA is concerned, we already made the successful effort to generalize the problem that Ansaldo STS posed to our attention, in order to exploit the solution we developed in situations other than the specific Ansaldo STS one. If we will be able to prove that the FYPA protocol is suited to manage even more MARA problems, we could move another step further and design

a **platform for developing negotiation protocols**, starting from the FYPA one. This platform could help users to develop new negotiation protocols step by step, starting from the definition of the model (entities, resources and their dependencies) and moving on with the definition of the rules regulating the interaction (specifying the time-outs, the priorities of entities...) and, for example, those to calculate the alternative allocations.

The next extension would be to integrate this **platform for developing negotiation protocols** into the DCaseLP rapid prototyping framework, thus resulting into an "Enhanced DCaseLP", in order to allow developers to specify rules regulating the agents behavior using a logical language such as tuProlog.

Finally, as proposed in [26], a further extension to our work could be to integrate the verification capabilities offered by Concurrent MetateM [56] into the "Enhanced DCaseLP" framework.

Concurrent MetateM is a multi-agent language in which each agent is programmed using a set of (augmented) temporal logic specifications of the behavior it should exhibit. These specifications are directly executed in order to generate the behavior of the agent. As a result, there is no risk of invalidating the logic as it can happen with systems where logical specification must first be translated to a lower level implementation. In order to verify MASs using temporal logic, agents should be specified in the Concurrent MetateM language instead of in natural language and then directly programmed in JADE, as it happens now, or in tuProlog, since DCaseLP already provides a seamless tuProlog-JADE interface. We are exploring the use of the Prometheus Design Tool for MASs [8] as a suitable means for specifying agents and their interactions in an high-level, declarative graphical language, and then translating this specification into both a Concurrent MetateM one for verification, and into JADE agents for simulation. In [34] a similar translation problem has already been successfully faced and we will ground our work upon it.

As far as our proposal to integrate ontologies in MASs is concerned, our research group has already extended the Ontology Agent, as described in Section 8.7. The "Enhanced DCaseLP" will take advantage of these results without any further effort since the extended Ontology Agent, being developed in JADE, will be able to be integrated into it for free.

Bibliography

[1] *AgentFly: reference homepage*. http://agents.felk.cvut.cz/projects/agentfly/.

[2] *AgentService: reference homepage*. http://www.agentservice.it/.

[3] *BeanGenerator: reference homepage*. http://hcs.science.uva.nl/usr/aart/beangenerator/index25.html.

[4] *JENA: Reference homepage*. http://jena.sourceforge.net/.

[5] *OKBC: reference homepage*. http://www.ai.sri.com/~okbc/.

[6] *OQL: from ODBMS page, a description of the OQL language*. http://www.odbms.org/download/001.04%20Ullman%20CS145%20ODL%20OQL%20Fall%202004.pdf.

[7] *OWL: reference homepage*. http://www.w3.org/TR/owl-features/.

[8] *Prometheus Design Tool for MASs: Reference homepage*. http://www.cs.rmit.edu.au/agents/pdt/.

[9] *PROTÉGÉ: reference homepage*. http://protege.stanford.edu/.

[10] *The Research of Intelligent Negotiation Agent- Application for B2C E-commerce*, volume 1, 2009.

[11] M. R. Adler, A. B. Davis, R. Weihmayer, and R. W. Worrest. Distributed artificial intelligence: vol. 2. chapter Conflict resolution strategies for nonhierarchical distributed agents, pages 139–161. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[12] Agent Technology Center (ATG). *AGLOBE: Reference homepage*. http://agents.felk.cvut.cz/aglobe.

[13] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[14] G. Agha, P. Wegner, and A. Yonezawa, editors. *Research directions in concurrent object-oriented programming*. MIT Press, Cambridge, MA, USA, 1993.

[15] A. Agogino and K. Tumer. Regulating air traffic flow with coupled agents. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 535–542, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[16] J. M. Alberola, J. M. Such, A. Garcia-Fornes, A. Espinosa, and V. Botti. A performance evaluation of three multiagent platforms. *Artif. Intell. Rev.*, 34:145–176, August 2010.

[17] American National Standard. KIF Knowledge Interchange Format. dpANS NCITS.T2/98-004, 1998. `http://logic.stanford.edu/kif/dpans.html`.

[18] M. Balduccini and M. Gelfond. Diagnostic reasoning with a-prolog. *Theory Pract. Log. Program.*, 3(4):425–461, 2003.

[19] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multi agent systems, 1995.

[20] J. Bates. The role of emotion in believable agents. *Commun. ACM*, 37:122–125, July 1994.

[21] K. D. Bilimoria, B. Sridhar, G. B. Chatterji, K. S. Shethand, and S. R. Grabbe. Future atm concepts evaluation tool. In *Air Traffic Control Quarterly , 9(1)*, 2001.

[22] M. Boman, P. Davidsson, N. Skarmeas, K. Clark, and R. Gustavsson. Energy saving and added customer value in intelligent buildings. *Building*, 1:505–516, 1998.

[23] G. Booch. Object-oriented analysis and design, second edition. *Reading, MA*, 1994.

[24] E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *Conference on Empirical Methods in Natural Language Processing, EMNLP 2002, Proceedings*, 2002.

[25] D. Briola, A. Locoro, and V. Mascardi. Ontology agents in fipa-compliant platforms: a survey and a new proposal. In *From Objects to Agents Workshop, WOA 2008, Proceedings*, 2008.

[26] D. Briola, M. Martelli, and V. Mascardi. Specification, simulation and verification of negotiation protocols in a unified agent-based framework (extended abstract). In *ICTCS 2010: 12th Italian Conference on Theoretical Computer Science*, 2010.

[27] D. Briola and V. Mascardi. Integrating a fipa-compliant ontology agent within a fipa-compliant framework. Technical Report DISI-TR-07-08, University of Genoa, Italy, 2007.

[28] D. Briola and V. Mascardi. Design and implementation of a netlogo interface for the standalone fypa system. In G. Fortino, A. Garro, L. Palopoli, W. Russo, and G. Spezzano, editors, *12th Workshop on Objects and Agents (WOA 2011), Proceedings*, volume Vol-741, pages 41–50. CEUR, 2011.

[29] D. Briola and V. Mascardi. Multi agent resource allocation: a comparison of five negotiation protocols. In G. Fortino, A. Garro, L. Palopoli, W. Russo, and G. Spezzano, editors, *12th Workshop on Objects and Agents (WOA 2011), Proceedings*, volume Vol-741, pages 95–104. CEUR, 2011.

[30] D. Briola, V. Mascardi, and M. Martelli. Intelligent agents that monitor, diagnose and solve problems: Two success stories of industry-university collaboration. In *Journal of Information Assurance and Security*, 2009.

[31] D. Briola, V. Mascardi, M. Martelli, G. Arecco, R. Caccia, and C. Milani. A prolog-based mas for railway signalling monitoring: Implementation and experiments. In *From Objects to Agents Workshop, WOA 2008, Proceedings*, 2008.

[32] D. Briola, V. Mascardi, M. Martelli, R. Caccia, and C. Milani. Dynamic resource allocation in a mas: A case study from the industry. In *From Objects to Agents Workshop, WOA 2009, Proceedings*, 2009.

[33] N. Carver and V. Lesser. A new framework for sensor interpretation: Planning to resolve sources of uncertainty. In *In Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 724–731, 1991.

[34] G. Casella and V. Mascardi. West2east; exploiting web service technologies to engineer agent-based software. *Int. J. Agent-Oriented Softw. Eng.*, 1:396–434, December 2007.

[35] N. Casellas, M. Blázquez, A. Kiryakov, P. Casanovas, M. Poblet, and R. Benjamins. OPJK into PROTON: Legal domain ontology integration into an upper-level ontology. In R. Meersman and et al., editors, *WORM 2005, 3rd International Workshop on Regulatory Ontologies, Proceedings*, volume 3762 of *LNCS*, pages 846–855. Springer, 2005.

[36] CDR, Stanford University. *JATLite Reference homepage*. `http://www-cdr.stanford.edu/ProcessLink/papers/JATL.html`.

[37] The Center for Connected Learning (CCL), Northwestern University. *StarLogo: Reference homepage*. `http://ccl.northwestern.edu/netlogo/docs/`.

[38] D. Cockburn and N. R. Jennings. Archon: A distributed artificial intelligence system for industrial applications, 1995.

[39] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42:213–261, March 1990.

[40] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–100, 1983.

[41] K. Decker. Designing behaviors for information agents. In *In Proceedings of the 1st Intl. Conf. on Autonomous Agents*, pages 404–412. ACM Press, 1997.

[42] S. A. DeLoach. Analysis and design using MaSE and agentTool. In *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*. Miami University, Oxford, Ohio, 2001.

[43] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 Mar. 2001. Proceedings.

[44] F. Dignum and C. Sierra, editors. *Agent Mediated Electronic Commerce, The European AgentLink Perspective*, volume 1991 of *Lecture Notes in Computer Science*. Springer, 2001.

[45] H. H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In A. B. Chaudhri, M. Jeckle, E. Rahm, and R. Unland, editors, *Proc. of NODe 2002*, pages 221–237. Springer, 2002.

[46] E. H. Durfee and J. S. Rosenschein. Distributed problem solving and multi-agent systems: Comparisons and examples. In *Proc. 13th Intl Distributed Artificial Intelligence Workshop*, pages 94–104, 1994.

[47] T. Eiter and V. Mascardi. Comparing environments for developing software agents. *AI Commun.*, 15(4):169–197, 2002.

[48] R. C. Erdur and O. Dikenelli. A fipa-compliant agent framework with an extra layer for ontology dependent reusable behaviour. In S. Berlin/Heidelberg, editor, *Advances in Information Systems: Second International Conference, ADVIS 2002, Proceedings*, volume 2457, pages 283–292, 2002.

[49] M. Esteva, J. A. Rodríguez-Aguilar, C. Sierra, P. Garcia, and J. L. Arcos. On the formal specifications of electronic institutions. In *AgentLink*, pages 126–147, 2001.

[50] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.

[51] T. W. Finin, R. Fritzson, D. P. McKay, and R. McEntire. KQML as an agent communication language. In *3rd International Conference on Information and Knowledge Management, CIKM'94, Proceedings*, pages 456–463. ACM, 1994.

[52] FIPA. *FIPA Ontology Service Specification*, 2001. http://www.fipa.org/specs/fipa00086/XC00086D.pdf.

[53] FIPA. *FIPA-RDF specification*, 2001. http://www.fipa.org/specs/fipa00011/XC00011B.html.

[54] FIPA. *FIPA ACL message specification*, 2002. http://www.fipa.org/specs/fipa00061/SC00061G.html.

[55] FIPA. *FIPA: The Foundation for Intelligent Physical Agents*, 2005. http://www.fipa.org/.

[56] M. Fisher. A survey of concurrent metatem - the language and its applications. In *Proceedings of the First International Conference on Temporal Logic*, ICTL '94, pages 480–505, London, UK, 1994. Springer-Verlag.

[57] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening ontologies with DOLCE. In A. Gómez-Pérez and V. R. Benjamins, editors, *EKAW, 13th International Conference, Proceedings*, volume 2473 of *LNCS*, pages 166–181. Springer, 2002.

[58] Garneau and Delisle. A new general, flexible and java-based software development tool for multiagent systems. In X. Yu and B. Ram, editors, *Teaching Intelligent Agents to Industrial Engineering Majors*. 2003.

[59] George Mason University's Evolutionary Computation Laboratory and the GMU Center for Social Complexity. *MASON: Reference page*. http://www.cs.gmu.edu/~eclab/projects/mason/.

[60] J. Gomez-Sanz and J. Pavon. Agent oriented software engineering with INGENIAS. In V. Marík, J. P. Müller, and M. Pechoucek, editors, *3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings*, volume 2691 of *LNCS*, pages 394–403. Springer, 2003.

[61] P. Grenon, B. Smith, and L. Goldberg. Biodynamic ontology: Applying BFO in the biomedical domain. In D. M. Pisanelli, editor, *Ontologies in Medicine*, volume 102 of *Studies in Health Technology and Informatics*, pages 20–38. IOS Press, 2004.

[62] B. Grosz and C. Sidner. Plans for discourse. In *Intentions in Comm.*, pages 417–444, 1990.

[63] T. Gruber. *Ontology, to appear in the Encyclopedia of Database Systems, Ling Liu and M. Tamer Ozsu (Eds.), Springer-Verlag, 2008.*

[64] H. Herre, B. Heller, P. Burek, R. Hoehndorf, F. Loebe, and H. Michalek. General formal ontology (GFO): A foundational ontology integrating objects and processes. part i: Basic principles. Technical report, Research Group Ontologies in Medicine (Onto-Med), University of Leipzig, 2006. Version 1.0, Onto-Med Report Nr. 8, 01.07.2006.

[65] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19:281–316, December 2004.

[66] B. Horling, R. Mailler, and V. Lesser. Farm: A scalable environment for multi-agent development and evaluation. In A. G. C. Lucena, J. C. A. Romanovsky, and P. Alencar, editors, *Advances in Software Engineering for Multi-Agent Systems*, pages 220–237. Springer-Verlag, Berlin, February 2004.

[67] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*. Montreal, Canada, 2001.

[68] B. Hubermanand and S. Clearwater. A multi-agent system for controlling building environments. In *Proceedings of the 1st International Conference on Multiagent Systems, 1995 June 12-14; San Francisco, CA*, volume 1, pages 171–176, 1995.

[69] IIIA. *EI survey, by IIIA*. `http://www.iiia.csic.es/~jsabater/SLIE-web/EI.html`.

[70] T. Ito, M. Zhang, V. Robu, S. Fatima, and T. Matsuo, editors. *Advances in Agent-Based Complex Automated Negotiations*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[71] N. R. Jennings. *Cooperation in industrial multi-agent systems*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1994.

[72] N. R. Jennings, E. H. Mamdani, J. M. Corera, I. Laresgoiti, F. Perriollat, P. Skarek, and L. Zsolt Varga. Using Archon to develop real-world DAI applications, part 1. *IEEE Expert*, 11(6):64–70, 1996.

[73] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[74] T. Knabe, M. Schillo, and K. Fischer. Improvements to the fipa contract net protocol for performance increase and cascading applications. In *In International Workshop for Multi-Agent Interoperability at the German Conference on AI (KI-2002*, 2002.

[75] R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[76] S. Lander and V. Lesser. Sharing Meta-Information to Guide Cooperative Search Among Heterogeneous Reusable Agents. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):193–208, January 1997.

[77] C. Leckie, R. Senjen, B. Ward, and M. Zhao. Communication and coordination for intelligent fault diagnosis agents. In *8th IFIP/IEEE International Workshop for Distributed Systems Operations and Management, DSOM'97, Proceedings*, pages 280–291, 1997.

[78] D. Lenat and R. Guha. *Building large knowledge-based systems*. Addison Wesley, 1990.

[79] V. R. Lesser. Cooperative multiagent systems: A personal view of the state of the art. *IEEE Transactions on Knowledge and Data Engineering*, 11:133–142, 1999.

[80] L. Li. Agent-based ontology management towards interoperability. Master's thesis, Swinburne University of Technology, 2005.

[81] L. Li, B. Wu, and Y. Yang. Agent-based ontology integration for ontology-based application. In *Proc. of Australasian Ontology Workshop (AOW 2005), the 18th Australian Joint Conference on Artificial Intelligence, Conferences in Research and Practice in Information Technology (CRPIT) series by Australian Computer Society, Vol 58*, pages 53–59, 2005.

[82] D. Lukose and Z. Shi, editors. *Multi-Agent Systems for Society: 8th Pacific Rim International Workshop on Multi-Agents, PRIMA 2005, Kuala Lumpur, Malaysia, September 26-28, 2005, Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 2009.

[83] A. P. Macedo. Metodologias de Negociação em Sistemas Multi-Agentes para Empresas Virtuais. Master's thesis, Faculdade de Engenharia, Universidade do Porto, 2000.

[84] P. Maes. Agents that reduce work and information overload. *Commun. ACM*, 37:30–40, July 1994.

[85] R. Mailler, V. Lesser, and B. Horling. Cooperative negotiation for soft real-time distributed resource allocation. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 576–583, New York, NY, USA, 2003. ACM.

[86] A. Malucelli and E. C. Oliveira. Ontology-services to facilitate agents' interoperability. In *PRIMA*, pages 170–181, 2003.

[87] V. Mascardi, D. Briola, M. Martelli, R. Caccia, and C. Milani. Monitoring and diagnosing railway signalling with logic-based distributed agents. In E. Corchado and R. Zunino, editors, *International Workshop on Computational Intelligence in Security for Information Systems, CISIS'08, Proceedings*, Advances in Soft Computing Series. Springer-Verlag, 2008.

[88] V. Mascardi, A. Locoro, and P. Rosso. Automatic ontology matching via upper ontologies: A systematic evaluation. *IEEE Trans. Knowl. Data Eng.*, 22(5):609–623, 2010.

[89] V. Mascardi, M. Martelli, and I. Gungui. DCaseLP: a prototyping environment for multi-language agent systems. In M. Dastani, A. E.-F. Seghrouchni, J. Leite, and P. Torroni, editors, *In Proceedings of the First Workshop on LAnguages, methodologies and Development tools for multi-agent systemS, LADS'007 Post-proceedings*, volume 5118 of *LNCS*, pages 139–155. Springer-Verlag, 2008.

[90] C. L. Mason and R. R. Johnson. Distributed artificial intelligence: vol. 2. chapter DATMS: a framework for distributed assumption based reasoning, pages 293–317. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[91] M. Menken. Jess tutorial. Vrije Universiteit, Amsterdam, The Netherlands, DEcember24 2002.

[92] B. A. Miller. The autonomic computing edge: Keeping in touch with touchpoints.

[93] F. Mulattieri. Bachelor thesis: Progettazione ed implementazione di un ontology agent. Master's thesis, DISI, University of Genova, Italy, 2010.

[94] D. E. Neiman, D. W. Hildum, V. R. Lesser, and T. W. Sandholm. Exploiting meta-level information in a distributed scheduling system. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, AAAI '94, pages 394–400, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[95] I. Niles and A. Pease. Towards a standard upper ontology. In C. Welty and B. Smith, editors, *FOIS 2001, 2nd International Conference on Formal Ontology in Information Systems, Proceedings*, pages 2–9. ACM Press, 2001.

[96] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

[97] T. Oates, M. V. N. Prasad, and V. R. Lesser. Cooperative information-gathering: a distributed problem-solving approach. *IEE Proceedings - Software*.

[98] M. Obitko and V. Snáěl. Ontology repository in multi-agent system. In M. H. Hamza, editor, *Artificial Intelligence and Applications, AIA 2004, Proceedings*, 2004.

[99] H. Parunak. Manufacturing experience with the contract net. *Distributed Artificial Intelligence*, pages 285–310, 1987.

[100] A. Pena, H. Sossa, and F. Gutierrez. Web-services based ontology agent. In *Distributed Frameworks for Multimedia Applications, 2006. The 2nd International Conference on*, pages 1–8, 2006.

[101] O. Purwin, R. D'Andrea, and J.-W. Lee. Theory and implementation of path planning by negotiation for decentralized agents. *Robot. Auton. Syst.*, 56(5):422–436, 2008.

[102] A. S. Rao and M. P. Georgeff. Modeling rational agents within a bdi-architecture. In *KR'91*, pages 473–484, 1991.

[103] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *KR*, pages 439–449, 1992.

[104] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model and technology for developing agent environments in MAS. In M. Dastani, A. E. Fallah-Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007*, volume 4908 of *LNCS*. Springer, 2008.

[105] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[106] M. Schroeder, I. de Almeida Móra, and L. Moniz Pereira. A deliberative and reactive diagnosis agent based on logic programming. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *4th International Workshop on Agent Theories, Architectures, and Languages, ATAL'97, Proceedings*, volume 1365 of *LNCS*, pages 293–307. Springer, 1998.

[107] G. S. Semmel, S. R. Davis, K. W. Leucht, D. A. Rowe, K. E. Smith, and L. Boloni. Space shuttle ground processing with monitoring agents. *IEEE Intelligent Systems*, 21(1):68–73, 2006.

[108] Y. Shoham. Agent-orientated programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[109] M. P. Singh. Towards a formal theory of communication for multi-agent systems. In *In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91*, pages 69–74. Morgan Kaufmann, 1991.

[110] M. P. Singh. *The Practical Handbook of Internet Computing*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[111] R. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, Series C-29(12):1104–1113, 1980.

[112] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing, 1999.

[113] G. Stoilos, G. B. Stamou, and S. D. Kollias. A string metric for ontology alignment. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *4th International Semantic Web Conference, ISWC 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 624–637. Springer, 2005.

[114] T. Sugawara and K. Murakami. A Multiagent Diagnostic System for Internetwork Problems. *Proceedings of INET'92*, January 1992.

[115] H. Suguri, E. Kodama, M. Miyazaki, H. Nunokawa, and S. Noguchi. Implementation of FIPA Ontology Service. In *Workshop on Ontologies in Agent Systems, Proceedings*, 2001.

[116] Swarm Development Group. *SWARM: Reference page*. `http://www.swarm.org/index.php/Swarm_main_page`.

[117] K. Sycara, S. Roth, N. Sadeh, and M. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1446–1461, 1991.

[118] M. Z. T. Ito, H. Hattori and T. Matsuo, editors. *Rational, Robust, and Secure Negotiations in Multi-Agent Systems Rational, Robust, and Secure Negotiations in Multi-Agent Systems*. Springer-Verlag, 2008.

[119] V. R. S. F. T. M. T. Ito, M. Zhang and H. Yamaki, editors. *Innovations in Agent-Based Complex Automated Negotiations*. Springer-Verlag, 1st edition, 2011.

[120] TILAB. *JADE: Reference homepage*. `http://jade.tilab.com/`.

[121] TILAB. *JADE: Tutorial on ontologies*. `http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf`.

[122] University College Dublin. *Agent Factory: Reference homepage*. `http://www.agentfactory.com/index.php/FAQ`.

[123] Uri Wilensky, at the Center for Connected Learning (CCL), Northwestern University. *Netlogo: Reference homepage*. `http://ccl.northwestern.edu/netlogo/`.

[124] U.S. Congress, Office of Technology Assessment. *Electronic Enterprises: Looking to the Future*. 1994.

[125] C. Vecchiola, A. Grosso, and A. Boccalatte. Integrating ontology support within AgentService. In *Seventh From Objects to Agents Workshop, WOA 2006, Proceedings*, 2006.

[126] D. Šišlák, J. Samek, and M. Pěchouček. Decentralized algorithms for collision avoidance in airspace. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 543–550, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[127] W3C. *RDQL specification*, 2004. `http://www.w3.org/Submission/RDQL/`.

[128] W3C. *SPARQL specification*, 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[129] R. Wei and C. Yongcan. *Distributed Coordination of Multi-agent Networks Distributed Coordination of Multi-agent Networks*. Springer Publishing Company, 1st edition, 2011.

[130] R. Weihmayer and R. Brandau. A distributed ai architecture for customer network control. In *Proceedings of GLOBECOM, 1990 December; San Diego*, 1990.

[131] T. Weihmayer and M. Tan. Modeling cooperative agents for customer network control using planning and agent-oriented programming. In *IEEE Global Telecommunications Conference, Globecom'92, Proceedings*, pages 537–543. IEEE, 1992.

[132] M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[133] M. P. Wellman and P. R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24(3-4):115–125, 1998.

[134] Wikipedia. Upper ontology – Wikipedia, the Free Encyclopedia, 2008. [Online; accessed 30-March-2008].

[135] F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Journal of Autonomous Agents and Multi-Agent Systems*, (3):253–283, 2004.

Appendix A

Simplified code of Nodo Agent

In this appendix, and in the next one, we will describe the implementation of Resource and User Agents in the "Ansaldo FYPA" system. Note that later on we will call:

- the User Agent "Treno"
- the Resource Agent "Nodo"
- the User Agents Manager "MovimentiIndotti"
- the Resource Agents Manager "PrenotazioniStazione"
- the Paths Manager "PercorsiAlternativi"

reflecting the name of classes and entities in the real application.

The original code used Italian words for methods and variables, and comments were in Italian too: in this thesis we report a simplified code of the agents with translated comments from Italian to English (to let the reader understand the reported code) but, to avoid inserting inconsistencies with respect to the original code and among the different pieces of simplified code, we kept the names of methods and variables as they were.

We report in this first appendix the simplified JADE code of the Nodo Agent, divided in main methods.

In this simplified code (as in the other Appendixes) we avoid reporting the code managing the creation and the running of threads: every new behavior added to the agents, except the first Cyclic one created in the setup, are usually created as Threaded Behaviors, as described in Section 2.3.2.5.

Later on we report the global variables and the setup method of Nodo agent.

```
Global variables of Nodo Agent

// structures to manage arcs and neighbors
  private List elencoItinerari_;
  private Vicino lista_vicini_[];
  private int numeroVicini_;

  // data describing the resource
  private boolean flagNodoIntermedio_;
  private boolean flagNodoTerminale_;
  private boolean flagNodoSosta_;
  private int idNodo_;

  // these Template are used to understand from which type of agent the message come from
  MessageTemplate template_tipo_ = MessageTemplate.MatchSender("Nodo_*");
  MessageTemplate templatePR_ = MessageTemplate.MatchSender("PrenotazioniStazione");

  // time-out for the reservation requests
  private long delta_;

  // lists of Train requests/reservations
  private Vector lista_prenotazioni_;
  private Vector lista_prenotazioni_rich_;
  private Vector lista_prenotazioni_att_;
  private Vector lista_prenotazioni_occ_;

  // global variables
  private long primo_da_ = -1;
  private long primo_a_ = -1;
  private int pri_altro_treno_ = -1;
  private int sarebbe_it_ = -1;
  private String altro_treno_ = "";
  private boolean altra_prenotaz_discutibile_=false;
  private List incompatibilita_;
  private long deltaSincronizzazione_;

protected void setup(){

    // milliseconds
    delta_ = 4000;

// reading the starting parameters
    Object[] args = getArguments();

// in class nodo there are all the information regarding the resource
    nodo det = (nodo)args[0];
    idNodo_ = det.getIdNodo();
    nomeNodo_ = "Nodo_" + det.getNomeNodo();
    flagNodoIntermedio_ = det.getFlagNodoIntermedio();
    flagNodoTerminale_ = det.getFlagNodoTerminale();
    flagNodoSosta_ = det.getFlagNodoSosta();
    elencoItinerari_ = det.getElencoItinerari();
    incompatibilita_ = (List)args[1];

    // read the Zero-Time
deltaSincronizzazione_=args[2];

  // this method will select from the complete list of arcs
  // only those concerning the Node and will fill the
  // lists with all the information about neighbors, arcs and incompatibilities
  // (stored in object of class Vicino).
```

```
 // We do not report this method because too complex, also using a big set of classes,
 // and it is not interesting at this level of detail.
   setNodiVicini(elencoItinerari_);

   lista_prenotazioni_ = new Vector();
   lista_prenotazioni_rich_ = new Vector();
   lista_prenotazioni_att_ = new Vector();
   lista_prenotazioni_occ_ = new Vector();

   controlla_timer_ = new TickerBehavior(this,2000) {
     // Every 2000 millisecond the agent will call the controlla_richieste method
     protected void onTick(){
       // in this method the agent will search for old reservations/requests and will delete them
       controlla_richieste();
     }
   };


   // Add the CyclicBehavior to get messages

   addBehavior(new CyclicBehavior(this) {
     public void action(){

         // search for messages from Resource Agents Manager
         msg = myAgent.receive(templatePR_);
         if(msg != null){

// this methods manages messages from PrenotazioniStazione
// (which sends the out of order status)
            manage_out_of_order(msg);

         }else{

             // search for messages from other Nodo agents
             msg = myAgent.receive(template_tipo_);
             if(msg != null){

               manage_nodo(msg);

             }else{

               // search for messages from Treno agents
               msg = myAgent.receive();
               if(msg != null){

                 manage_treno(msg);

               }else{
                 block();
               }
             }
          }
       }
   });
 }
```

To understand the next code we report the code of the main classes used to represent a reservation ("Prenotazione_Nodo" class) and a neighbor ("Vicino" class). The structure of classes representing neighbors is very complex and it is not interesting, in this thesis, to give too much

details because it would require too technical information. So we report these classes without describing all the structures or nested classes but only describing at high level their role and their main methods.

```
Class Prenotazione_Nodo

private long Da_; // From
  private long A_; // to
  private boolean Confermata_; // confirmed
  private boolean Discutibile_; // disputable
  private String Treno_; // train
  private int Priorita_; // train's priority
  private int Num_; // number of the reservation
  private String Nodo_provenienza_Treno_; // Nodo from the train comes
  private int Itinerario_assegnato_; // arc to use
  private long Ora_; // time when the reservation has been received

  /** Creates a new instance of Prenotazione_Nodo */
  public Prenotazione_Nodo(int num,
                           long da,
                           long a,
                           String treno,
                           boolean disc,
                           boolean conf,
                           int itinerario,
                           String nodo,
                           long ora,
                           int pri){
    Da_ = da; // from
    A_ = a; // to
    Treno_ = treno; // train
    Confermata_ = conf; // confirmed
    Discutibile_ = disc; // disputable
    Num_ = num; // number
    Itinerario_assegnato_ = itinerario; // arc
    Nodo_provenienza_Treno_ = nodo; // from node
    Ora_ = ora; // time of the creation of the object
    Priorita_ = pri; // priority of the train
  }

  // Set and Get methods for all fields
  // not reported



Class Vicino

  private String nomeVicino_;
  private Itinerario lista_itinerari_[];

  // this is the traveling time of the arcs
  private long tempo_percorrenza_ = 40;

  private AgenteNodo parent_;

// when the Vicino is created, it has a list of all arcs and nodes.
// this information has to be filtered and reorganized for every Vicino
```

```
  public Vicino(String nome,
                int[] itinerari,
                Vector[] nodi,
                AgenteNodo master){

    this.nomeVicino_ = nome;
    parent_=master;

// the class Itinerari manages the organization of arcs and incompatibility information.
// We do not report it.
    lista_itinerari_ = new Itinerario[itinerari.length];

    for(int i = 0;i < itinerari.length;i++){

      lista_itinerari_[i] = new Itinerario(itinerari[i],
                                           false,
                                           (Vector)nodi[i],
                                           tempo_percorrenza_,
                                           this);
    }
    // in the end in lista_itinerari_ there will be the list of only those arcs managed by this Vicino,
    // with all the information regarding incompatibilities

  }


// this method returns the number of one arc (from those managed by this Vicino)
// that is free in the interval da-a
  public int itinerario_libero(long da,long a, String treno, boolean locale, String nodo){
    Itinerario it_app;

    for(int m = 0;m < lista_itinerari_.length;m++){
      it_app = lista_itinerari_[m];

      // the method "libero" of class Itinerario will check if the arc is free in that interval
      if(it_app.libero(da,a,treno, locale, nodo)){
        return it_app.getNumero();
      }
    }
// -1 stands for "no free arcs available"
    return -1;
  }

// this method adds a reservation for the arc "num"
  public void inserisci_prenotazione_itinerario(
   int num,
                                            long da,
                                            long a,
                                            String treno,
                                            String nodo,
                                            String nodo_princ,
                                            boolean conf){


// NOTE THAT the real interval of reservation for an arc is only the time needed to pass through it:
// ("a = da + tempo_percorrenza_"  means  "to= from  + traveling time")
// except when the train is "rottura it", that means that the arc is out of order
// for all the specified time
    if(!treno.equals("rottura it")) a = da + tempo_percorrenza_;
```

```
Itinerario app;
    for(int m = 0;m < lista_itinerari_.length;m++){
      app = lista_itinerari_[m];
      if(app.getNumero() == num){
        app.inserisci_prenotaz_itinerario(da,
                                          a,
                                          treno,
                                          nodo,
                                          conf,
                                          nodo_princ);
        break;
      }
    }
  }


// this method deletes or updates (with respect to the value in the field "elimina")
// the reservations on the arc "num"
  public void elimina_modifica_prenotazione(int num,
                                        long da,
                                        long a,
                                        String nodo_pr,
                                        String nodo_chiamante,
                                        boolean conf,
                                        boolean elimina){
// not reported
  }

// it returns a list of the neighbors (object) interested in the arc "it", that is
// the list of Nodo agents that manage at least an arc incompatible with "it".
  public Vector get_nodi_interessati(int it){
    Itinerario app = new Itinerario();
    for(int m = 0;m < lista_itinerari_.length;m++){
      app = lista_itinerari_[m];
      if(app.getNumero() == it){
       // this method returns a list of objct of a class, not reported here,
       // that is used to represent nodes with entering arcs and incompatibilities
        return app.get_nodi_interessati();
      }
    }
    return new Vector();
  }

// it deletes the old reservations
  public void cacella_vecchie_pr(long deltasinc){
// not reported
}
```

Later on we report the simplified code of the method $manage\_nodo$.

```
private void gestisci_aggiornamenti_nodi(ACLMessage msg){
// message content format
// [(it1,nodoA),(it2,nodoB)...];From;To;Conf;Train;inf

// parse the String content:
// put in "con" variabe the list [(it1,nodoA),(it2,nodoB)...]
```

```
  int perf = msg.getPerformative();
  String con = msg.getContent();
  int x = con.indexOf("]");
  con = con.substring(0,x + 1);

  // parse the remaining part of the message
  String ncon = msg.getContent().substring(x + 2,msg.getContent().length());
  x = ncon.indexOf(";");
  long da = Long.parseLong(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  long a = Long.parseLong(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  boolean conf = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  String treno=ncon.substring(0,x);
  ncon = ncon.substring(x + 1,ncon.length());
  int it=Integer.parseInt(ncon);

  // Parse the list con
  if(con.charAt(1) != ']'){
    int num; x = 0;
    String nodo;
    while(true){
      con = con.substring(2,con.length());
      x = con.indexOf(",");
      num = Integer.parseInt(con.substring(0,x));
      con = con.substring(x + 1,con.length());
      x = con.indexOf(")");
      nodo = con.substring(0,x);
      con = con.substring(x + 1,con.length());
      // in the list there are the name of my Vicino (neighbors)
      Vicino app = new Vicino();
      int n = 0;
      for(n = 0;n < lista_vicini_.length;n++){
        app = lista_vicini_[n];
        if(app.getNomeVicino().equals(nodo)){
          // I found the correct Vicino
          break;
        }
      }

      // looking at the performative of the message
      // insert/confirm/cancel the reservation request
      if(INFORM || AGREE){
        app.inserisci_prenotazione_itinerario(num,da,a,treno,nodo,this.getLocalName(),conf);
      }else if(CONFIRM){
        app.elimina_modifica_prenotazione(num,da,a,nodo,this.getLocalName(),conf,false);
      }else if(CANCEL){
        app.elimina_modifica_prenotazione(num,da,a,nodo,this.getLocalName(),conf,true);
      }

      if(con.charAt(0) == ']'){
        break;
      }
    }
  }else{
    //  no arc
  }
}
```

Later on we report the simplified code of the method $manage\_treno$.

```
private void gestisci_messaggio(ACLMessage msg){
    boolean flagItinerarioLibero;
    int it = -1;

    String treno = msg.getSender().getLocalName();
    int perf = msg.getPerformative();
    String con = msg.getContent();
    int x = con.indexOf(";");
    int num = Integer.parseInt(con.substring(0,x));
    String ncon = con.substring(x + 1,con.length());
    x = ncon.indexOf(";");
    long da = Long.parseLong(ncon.substring(0,x));
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");
    long a = Long.parseLong(ncon.substring(0,x));
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");
    String da_nodo = ncon.substring(0,x);
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");
    boolean flagDiscutibile = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
    ncon = ncon.substring(x + 1,ncon.length());
    int pri = Integer.parseInt(ncon);

    ACLMessage reply = msg.createReply();

    if(perf == QUERY_IF){
      manage_QUERY_IF(reply, treno, num, da, a, da_nodo, flagDiscutibile);
    }
    else if(perf == CONFIRM){
     manage_CONFIRM(reply, treno, num, da, a, da_nodo, flagDiscutibile);
    }
    else if(perf == AGREE){
     manage_AGREE(reply, treno, num, da, a, da_nodo, flagDiscutibile);
    }
    else if(perf == CANCEL){
     elimina_prenotazione_completa(num,da,a,treno,flagDiscutibile,pri,lista_prenotazioni_);
    }

}
```

Later on we report the simplified code of the methods which manage the incoming messages from Trains.

At first the method $manage\_QUERY\_IF$ is reported.

```
private void manage_QUERY_IF(AclMessage reply, String treno, int num,
long da, long a, String da_nodo, boolean flagDiscutibile){

  if(flagDiscutibile){
    // check if free (no other confirmed reservation) and, if not,
    // memorize when the resource will be again free (save in global var primo_da_,primo_a_
    if(libero(da,a,lista_prenotazioni_,treno)){

// check if free (no other confirmed reservation) and, if not,
```

```
     // memorize when the resource will be again free (save in global var primo_da_,primo_a_
      if(libero(da,a,lista_prenotazioni_rich_,treno)){

// check if the train is coming from me (I'm the first node of the path):
// in this case I can't find a path connecting me to me!
// otherwise, look for a free entering arc connection me to the da_nodo Nodo
        int n = 0;
        if(da_nodo.equals(this.getLocalName())){
          flagItinerarioLibero = true;
          it = 0;
        }else{
          Vicino app;
          flagItinerarioLibero = false;
          for(n = 0;n < lista_vicini_.length;n++){
            app = lista_vicini_[n];
            if(app.getNomeVicino().equals(da_nodo)){
// find, if any, a free arc
              it = app.itinerario_libero(da,a,treno,false,this.getLocalName());
              if(it != -1){
                flagItinerarioLibero = true;
              }
              break;
            }
          }
        }

        if(flagItinerarioLibero){
         // if a free arc exists
          // store this reservation request.
          // this method will also send the INFORM messages to the neighbors
          //interested in arc num
          inserisci_prenotaz_completa(num,da,a,treno,flagDiscutibile,false,it,
           da_nodo,lista_prenotazioni_rich_,pri);
          reply.setPerformative(ACLMessage.CONFIRM);
          // specify the it (arc) to be used
          reply.setContent(num + ";true;" + it);
          this.send(reply);
        }else{
          reply.setPerformative(ACLMessage.INFORM);
// all the -1 indicates that the nodo is not available (in this case because there are no free arcs)
          reply.setContent(num + ";false;-1;-1;-1;-1;-1");
          this.send(reply);
        }
      }else{ // other reservation requests (pending) exist
        inserisci_prenotazione(num,da,a,treno,flagDiscutibile,false,0,da_nodo,
         lista_prenotazioni_att_,pri);
      }
    }else{ // other already confirmed reservations exist
      reply.setPerformative(ACLMessage.INFORM);

      if(altra_prenotaz_discutibile_){
        // altra_prenotaz_discutibile_ specifies if the already
        // existing reservation is Disputable or not

        // if Disputable I will add in the message when I will free again
        reply.setContent(num + ";false;" + primo_da_ + ";" + primo_a_ + ";"
        + pri_altro_treno_ + ";" + altro_treno_ + ";" + sarebbe_it_);
        this.send(reply);
        inserisci_prenotazione(num,da,a,treno,flagDiscutibile,false,sarebbe_it_,
        da_nodo,lista_prenotazioni_occ_,pri);
      }
      else{
```

```
// if NOT Disputable I will add in the message when I will free again
// but I add that the priority of the other train is -1 (even if it is not true):
// in this way nobody can steal this reservation
        reply.setContent(num + ";false;" + primo_da_ + ";" + primo_a_ + ";-1;" +
        altro_treno_ + ";" + sarebbe_it_);

        this.send(reply);
        // do not add this reservation in any list because nobody can send me back an AGREE
      }
    }

  }else{ If the reservation request was Not Disputable

    if(da_nodo.equals(this.getLocalName())){
      flagItinerarioLibero = true;
      it = 0;
    }else{
      Vicino app;
      flagItinerarioLibero = false;
      for(int n = 0;n < lista_vicini_.length;n++){
        app = lista_vicini_[n];
        if(app.getNomeVicino().equals(da_nodo)){
          it = app.itinerario_libero(da,a,treno,true,this.getLocalName());
          if(it != -1){
            flagItinerarioLibero = true;
          }
          break;
        }
      }
    }

    if(flagItinerarioLibero){

// this method will check if the resource if free in that interval considering only
// already NOT Disputable reservation: Disputable reservation are not considered
      if(libero_disc(da,a,lista_prenotazioni_,treno)){
// this method will also send the INFORM messages to the neighbors interested in arc num
        inserisci_prenotaz_completa(num,da,a,treno,flagDiscutibile,false,
        it,da_nodo,lista_prenotazioni_rich_,pri);
        reply.setPerformative(ACLMessage.CONFIRM);
        reply.setContent(num + ";true;" + it);
        this.send(reply);
       }
      else{
        // If other NOT Disputable reservation existi than I can not accept this reservation
        reply.setPerformative(ACLMessage.INFORM);
// again, priority will be -1
        reply.setContent(num + ";false;" + primo_da_ + ";" + primo_a_ +
        ";-1;" + altro_treno_ + ";" + sarebbe_it_);
        this.send(reply);
      }

    }else{
      // no free arc
      reply.setPerformative(ACLMessage.INFORM);
      reply.setContent(num + ";false;-1;-1;-1;-1;-1");
      this.send(reply);
    }
  }
}
```

Subsequently the method $manage\_CONFIRM$ is reported.

```
private void manage_CONFIRM(AclMessage reply, String treno, int num,
long da, long a, String da_nodo, boolean flagDiscutibile){

  Prenotazione_Nodo el;
for(int i = 0;i < lista_prenotazioni_rich_.size();i++){
  el = (Prenotazione_Nodo)lista_prenotazioni_rich_.elementAt(i);
  // I need only the reservation number and the train to identify the reservation
  if(el.getNum() == num && (el.getTreno()).equals(treno)){
    if(el.isDiscutibile()){
      // if Disputable
      // delete the reservation request
      elimina_prenotazione(el.getNum(),
                           el.getDa(),
                           el.getA(),
                           el.getTreno(),
                           el.isDiscutibile(),
                           lista_prenotazioni_rich_);

      // and add the confirmed reservation to the correct list
      inserisci_prenotazione(el.getNum(),
                             el.getDa(),
                             el.getA(),
                             el.getTreno(),
                             el.isDiscutibile(),
                             true,
                             el.getItinerario_assegnato(),
                             el.getNodo_provenienza_Treno(),
                             lista_prenotazioni_,
                             el.getPriorita());
    }else{ // if Not disputable

      elimina_prenotazione(el.getNum(),
                           el.getDa(),
                           el.getA(),
                           el.getTreno(),
                           el.isDiscutibile(),
                           lista_prenotazioni_rich_);

      // in this method I will add the reservation to my list
      // but I will also inform the other trains, if any, that they lost their reservation
      // extract from this method:

      //    msg = new ACLMessage(ACLMessage.CANCEL);
      //    this is the train that had the resource that the new train is stealing
        //    msg.addReceiver(el.getTreno());
        //    primo_da_ +";" + primo_a_ + ";" + pri + ";" + treno
        //    refer to the information of the stealing train
        //    (from-to when I will free again, priority of the stealing train, its name)
        //    msg.setContent(el.getNum() + ";false;" + primo_da_ +";" + primo_a_ +
        //     ";" + pri + ";" + treno);
//    send(msg);

      elimina_prenotazione_completa(0,
                                    da,
                                    a,
                                    treno,
                                    flagDiscutibile,
                                    pri,
                                    lista_prenotazioni_);
```

```
      // and also delete the other reservation request that had a CONFIRM response
      elimina_prenotaz_completa2(0,
                                 da,
                                 a,
                                 treno,
                                 flagDiscutibile,
                                 lista_prenotazioni_rich_);

      // add the reservation to the list
      inserisci_prenotazione(num,
                             da,
                             a,
                             treno,
                             flagDiscutibile,
                             true,
                             el.getItinerario_assegnato(),
                             da_nodo,
                             lista_prenotazioni_,
                             pri);
  }
    // then, inform the neighbors of the new state of the arc
    Vicino app;
    flagItinerarioLibero = false;
    for(int n = 0;n < lista_vicini_.length;n++){
      app = lista_vicini_[n];
      if(app.getNomeVicino().equals(el.getNodo_provenienza_Treno())){
        app.elimina_modifica_prenotazione(el.getItinerario_assegnato(),
                                          el.getDa(),
                                          el.getA(),
                                          this.getLocalName(),
                                          this.getLocalName(),
                                          true,
                                          false);

        // ask the object Vicino the list of the interested neighbors to the arc assigned
        // and inform them
        Vector nodi_int = app.get_nodi_interessati(el.getItinerario_assegnato());
        ACLMessage msg2;
        for(int m = 0;m < nodi_int.size();m++){
         // this class represent one of the neighbors with incompatible arcs
          mioVicinoIncomp mv = ((mioVicinoIncomp)nodi_int.get(m));
          msg2 = new ACLMessage(ACLMessage.CONFIRM);
          msg2.addReceiver(new AID(mv.nome,false));
          msg2.setSender(this.getAID());
          // mv.get_elenco_itinerari() returns the list of interested arc with their origin node,
          // already formatted for the message
          // [(arc, FromNode),(arc2, FromNode2)...]
          msg2.setContent(mv.get_elenco_itinerari() + ";" + el.getDa() +
                  ";" + el.getA() + ";true;" + el.getTreno() + ";-1");
          send(msg2);
        }
        break;
      }
    }
// let's manage the reservation request waiting for an answer
    gestisci_richieste_appese(false,da,a,el.isDiscutibile(),false);
    return;
  }
}   // end for on the list
```

```
// if I arrive in this point of the code there was a problem with this
// reservation request: it has expired!!
// So I have to send a CANCEL to the train because it think that its reservation is confirmed!
reply.setPerformative(ACLMessage.CANCEL);
reply.setContent(num + ";false;-1;-1;-2");
this.send(reply);
}
```

Lastly the method $manage\_AGREE$ is reported.

```
private void manage_AGREE(AclMessage reply, String treno, int num,
long da, long a, String da_nodo, boolean flagDiscutibile){
  // this method is very similar to the Confirm one.
  // I have to search in other lists but the main idea is the same

Prenotazione_Nodo el;
// search for the previous reservation:
// in this case it will be among those I answered with an "occupied" (INFORM)
for(int i = 0;i < lista_prenotazioni_occ_.size();i++){
  el = (Prenotazione_Nodo)lista_prenotazioni_occ_.elementAt(i);
  if ((el.getNum() == num) && (el.getTreno()).equals(treno)){
// cancel the old reservation
    elimina_prenotazione(el.getNum(),
                         el.getDa(),
                         el.getA(),
                         el.getTreno(),
                         el.isDiscutibile(),
                         lista_prenotazioni_occ_);

    // cancel the other reservation in that interval and inform the other train
    elimina_prenotazione_completa(0,da,a,treno,flagDiscutibile,pri,lista_prenotazioni_);

    if(da_nodo.equals(this.getLocalName())){
      flagItinerarioLibero = true;
      it = 0;
    }else{
      Vicino app;
      flagItinerarioLibero = false;
      for(int n = 0;n < lista_vicini_.length;n++){
        app = lista_vicini_[n];
        if(app.getNomeVicino().equals(da_nodo)){
          it = app.itinerario_libero(da,a,treno,false,this.getLocalName());
          if(it != -1){
            flagItinerarioLibero = true;
          }
          break;
        }
      }
    }
    inserisci_prenotazione(num,
                           da,
                           a,
                           treno,
                           flagDiscutibile,
                           true,
                           it,
                           da_nodo,
                           lista_prenotazioni_,
                           pri);
    Vicino app;
```

```
    flagItinerarioLibero = false;
    for(int n = 0;n < lista_vicini_.length;n++){
      app = lista_vicini_[n];
      if(app.getNomeVicino().equals(el.getNodo_provenienza_Treno())){

        app.elimina_modifica_prenotazione(it,
                                    el.getDa(),
                                    el.getA(),
                                    this.getLocalName(),
                                    this.getLocalName(),
                                    true,
                                    false);

        Vector nodi_int = app.get_nodi_interessati(it);
        ACLMessage msg2;
        for(int m = 0;m < nodi_int.size();m++){
          mioVicinoIncomp mv = ((mioVicinoIncomp)nodi_int.get(m));
          msg2 = new ACLMessage(ACLMessage.AGREE);
          msg2.addReceiver(new AID(mv.nome,false));
          msg2.setSender(this.getAID());
          msg2.setContent(mv.get_elenco_itinerari() + ";" + el.getDa() +
                  ";" + el.getA() + ";true;" + el.getTreno() + ";-1");
          send(msg2);
        }
        break;
      }
    }

    gestisci_richieste_appese(false,da,a,el.isDiscutibile(),true);
    return;
  }   // end if==el.Num
}   // end for sulla lista

// if I am here the reservation has expired
reply.setPerformative(ACLMessage.CANCEL);
reply.setContent(num + ";false;-1;-1;-1;-10");
this.send(reply);
}
```

Later on we report the simplified code of the method $manage\_out\_of\_order$. Note that the message received from the PrenotazioniStazione (Resource Agents manager) will have this format:

data;from;to;it;fromNodo;NodoOut

- data: day of this out of order status
- From: Starting Time of out of order status
- To: Ending Time of out of order status
- it: arc (if different from -1) interested
- fromNodo: departing Nodo of the arc it
- NodoOut: boolean to specify if also the resource is out of order

The out of order status is managed as a non disputable reservation by a Train “rottura nodo” with priority -1.

```
private void manage_out_of_order(ACLMessage msg){

    String data;
    long da;
    long a;
    int it;
    String da_nodo;
    boolean rotto_nodo;

// by reference parameters
read_data(msg, data, da, a, it, da_nodo, rotto_nodo);
// populate the variables with the values in the messages

    // The out of order status is managed as a non disputable reservation.
    if(rotto_nodo){
        // also the resource is out of order

// delete all the already confirmed reservations interested by this out of order status
        elimina_prenotazioni_per_rottura(0,
                                         da,
                                         a,
                                         "rottura nodo",
                                         false,
                                         -1,
                                         lista_prenotazioni_);

// insert a new already confirmed not disputable reservation for the resource,
// made by a train "rottura nodo"
        inserisci_prenotazione(-1,
                               da,
                               a,
                               "rottura nodo",
                               false,
                               true,
                               0,
                               this.getLocalName(),
                               lista_prenotazioni_,
                               -1);
    }

    if(it!=-1){
        // if the arc is out of order

        // find the neighbor app that manages the arc
        Vicino app = new Vicino();
        int n = 0;
        for(n = 0;n < lista_vicini_.length;n++){
          app = lista_vicini_[n];
          if(app.getNomeVicino().equals(da_nodo)){
            break;
          }
        }

// add a new reservation on the arc
        app.inserisci_prenotazione_itinerario(it, da, a, "rottura it", this.getLocalName(),
        this.getLocalName(), true);
```

```
        // get the list of the other nodo agents interested by the update on arc "it"
        // and inform them of this new reservation with INFORM messages
        app.update_status(it, da, a, true, "rottura it;");


         elimina_risposte_attesa(it, da, a);

         for(int i=0; i<lista_prenotazioni_.size();i++){
         Prenotazione_Nodo el;
         el=(Prenotazione_Nodo)lista_prenotazioni_.get(i);
         if(el.getItinerario_assegnato()==it && el.getDa() < a && el.getA() > da){
// look for another arc that can be used insted of "it"

             int nuovo_it = app.itinerario_libero(el.getDa(),el.getA(),"rottura it",
             false,this.getLocalName());

             if(nuovo_it != -1){
                   // there is another free arc
                   // send a message to Train agents informing them that they have
                   //to change the arc from "it" too "nuovo_it"
                   sostituisci_it(el.getTreno(), el.getNum(), it, nuovo_it, app);

               }
               else{

// delete the reservation and inform the Train,
// and send it the future free time interval
                   lista_prenotazioni_.remove(i);
                   msg = new ACLMessage(ACLMessage.CANCEL);
                   msg.addReceiver(new AID(el.getTreno(),false));
                   msg.setSender(this.getAID());
                   msg.setContent(el.getNum() + ";false;" + (a +1) + ";" + ((a +1) +
                   (el.getA()-el.getDa())) + ";-1;" + "rottura it");

                   send(msg);

                   i--;
               }

         }
        }

    }

  }
```

Appendix B

Simplified code of Treno Agent

Later on we report the simplified JADE code of the Treno Agent, starting with its global variables and $setup()$ method.

```
Global variables of Treno Agent

// structures and classes to memorize the initial data
// including the original path
  private Dati datiTreno_;
  private List TDM_;
  private String nomeTreno_;
  private String chiaveTreno_;
  private int priorita_;
  // delta_ is the time-out for the reservation requests
  private long delta_;

// template to understand the sender of a message
  private MessageTemplate template_nodo_ = MessageTemplate.MatchSender("nodo_*");
  private MessageTemplate templatePA_ = MessageTemplate.MatchSender("PercorsiAlternativi");

  private Behavior controlla_timer_;
  private Behavior controlla_bloccato_;

  // Zero time
  private long deltaSincronizzazione_;

  // how much delay I can suffer
  private long max_ritardo_acc_;

  // the current total delay
  private long ritardo_accumulato_;

  // if true I must accept any delay
  private boolean accetta_comunque_ritardo;

  // the first stores if I cannot move (there are no paths)
  // the second how long I will wait before starting again searching for a path
  private boolean sono_bloccato;
  private long delta_stop_;
```

```
  // it stores if the original path was interrupted (some node out of order) or not
  private boolean percorso_originale_interrotto_;

  // index to know at what step of the current path I am
  private int dove_sono_;

  // original path, never changed
  private Vector lista_passi_originale_;

  // temporary list of the current path
  private Vector lista_passi_aggiornata_;

  // used when I ask the PA
  private Vector lista_passi_prima_di_PA;

  // it is the current path I am trying to reserve
  private Vector lista_passi_corrente_;

  private Vector lista_prenotazioni_richieste_;
  private Vector lista_prenotazioni_accettate_;
  private Vector lista_prenotazioni_non_accettate_;
  private Vector lista_prenotazioni_confermate_;

// true if I have already asked for alternative path starting from a Nodo
  private boolean inviata_richiesta_PA_;

  // which alternative path I'm considering, choosen from the
  // list with all the possible alternative paths
  private int num_percorso_;
  private Vector lista_percorsi_alternativi_;

  // used to manage the alternative paths
  private boolean flag_percorso_alternativo_;
  private boolean flag_percorso_alternativo_confermato_;
  private boolean flag_percorso_alternativo_suggerito_;

  // true if I'm till searching for a path
  private boolean cerco_percorso_;

  // used to count the Nodo answers
  private int passi_confermati_;
  private int passi_risposti_;
  private int prgPrenotazioni_;

// the name of the other, if any, Train I stole a resource from
  private String treno_concorrente_;


protected void setup(){

// when I start, I'm searching for a path, I'm out of the station,
// I have not yet contacted the PA and I have all empty lists
    dove_sono_=-1;
    flag_percorso_alternativo_=false;
    flag_percorso_alternativo_confermato_=false;
    flag_percorso_alternativo_suggerito_=false;
    percorso_originale_interrotto_=false;
    cerco_percorso_ = true;
    passi_confermati_ = 0;
    passi_risposti_ = 0;
    prgPrenotazioni_ = 0;
    delta_ = 10000;
```

```
    delta_stop_=20000;
    max_ritardo_acc_ = 180000;
    ritardo_accumulato_ = 0;
    sono_bloccato=false;
    accetta_comunque_ritardo=false;
    lista_prenotazioni_confermate_ = new Vector();
    lista_prenotazioni_richieste_ = new Vector();
    lista_prenotazioni_accettate_ = new Vector();
    lista_passi_originale_ = new Vector();
    lista_prenotazioni_non_accettate_ = new Vector();
    lista_percorsi_alternativi_ = new Vector();
    lista_passi_aggiornata_ = new Vector();
    lista_passi_prima_di_PA=new Vector();
    lista_prenotazioni_vincolate_ = new Vector();
    inviata_richiesta_PA_ = false;
    num_percorso_ = 0;
    treno_concorrente_="";
    Object[] args = getArguments();

     // class Dettaglio stores my data
      Dettaglio det = (Dettaglio)args[0];
      datiTreno_ = det.getDatiTreno();

      // get the original path
      TDM_ = det.getTDM();
      nomeTreno_ = String.valueOf(datiTreno_.getIdTreno());
      priorita_=datiTreno_.getIdPriorita();
      chiaveTreno_ = datiTreno_.getChiaveTreno();

// zero time
      deltaSincronizzazione_= Long.parseLong(args[1]);

      // this method add object of type PassoTreno, which represent a step of the path,
      // in the list lista_passi_originale_, reading from TDM_
      setPercorso();

// this method will send the reservation requests to the Nodo agents
// that manage the resource in the list (path) passed as parameter
      prenota_percorso(lista_passi_originale_);

      // activete a CyclicBehavior to wait for the answer from Nodo agents
      attivaRicezioneMessaggi();
   }

public void attivaRicezioneMessaggi(){
  addBehavior(new CyclicBehavior(this) {
    public void action(){
        msg = myAgent.receive(templatePA_);
        if(msg != null){
          // answer from PA
          gestisci_risposta_PercorsiAlternativi(msg);
        }else{
          // message from nodo
          msg = myAgent.receive(template_nodo_);
          if(msg != null){
            gestisci_aggiornamenti_nodi(msg);
          }else{block();}
       }
  });
}
```

Later on the simplified code of methods *prenota_percorso(Vector path)* and *controlla_conferme()* is reported.

```
private void prenota_percorso(Vector percorso_da_richiedere){
  // class to describe a step in the path
  PassoTreno passoCorrente;

ACLMessage msg;
  String nodo_precedente = ((PassoTreno)(percorso_da_richiedere.elementAt(0))).getNodo();
  passi_confermati_ = 0;
  passi_risposti_ = 0;
  lista_passi_corrente_ = new Vector();
  for(int n = 0;n < percorso_da_richiedere.size();n++){
    passoCorrente = (PassoTreno)percorso_da_richiedere.elementAt(n);

    // add to the list each step pf the path,
    // storing the name of the nodo, if it is the first of the path,
    // when I leave the previous nodo, when I arrve on the nodo and when I leave it
    // and if the nodo is a "Stop node"
    lista_passi_corrente_.add(n,new PassoTreno(passoCorrente.getNodo(),
                                               passoCorrente.isFirstStep(),
                                               passoCorrente.getOraPartenzaNodoPrecedente(),
                                               passoCorrente.getOraArrivoNodo(),
                                               passoCorrente.getOraPartenzaNodo(),
                                               passoCorrente.isSosta()));

    // send a QUERY_IF message to the Nodo agent
    msg = new ACLMessage(ACLMessage.QUERY_IF);
    msg.addReceiver(new AID(passoCorrente.getNodo(),false));
    msg.setSender(this.getAID());

    // usually, the request for the first step is Non disputable
    boolean discutibile;
    if(nodo_precedente.equals(passoCorrente.getNodo())){
      discutibile = false;
    }else{
      discutibile = true;
    }

    // counter of the reservation requests sent
    prgPrenotazioni_++;

    // send the message to the Nodo
    msg.setContent(prgPrenotazioni_ + ";" + passoCorrente.getOraPartenzaNodoPrecedente()
            + ";" + passoCorrente.getOraPartenzaNodo() + ";" +
            nodo_precedente + ";" + discutibile + ";" + priorita_);
    send(msg);

    // add this reservation request to the list lista_prenotazioni_richieste_
    inserisci_prenotazione(prgPrenotazioni_,
                           passoCorrente.getNodo(),
                           0,   // arc: 0 stands for "to be decided"
                           passoCorrente.getOraPartenzaNodoPrecedente(),
                           passoCorrente.getOraArrivoNodo(),
                           passoCorrente.getOraPartenzaNodo(),
                           discutibile,
                           false,// it indicates if I am stealing a resource or not
                           lista_prenotazioni_richieste_);
    nodo_precedente = passoCorrente.getNodo();
  }
```

```
  // start a new WakerBehavior that will be activated ofter a predefined time-out, delta_.
  // That is, if the WakerBehavior starts (no one canceled it), it means that
  // I did not received all the answers from Nodo
  controlla_timer_ = new WakerBehavior(this,delta_) {
    protected void onWake(){
      controlla_conferme();
    }
  };
  addBehavior(controlla_timer_);
}


private void controlla_conferme(){
// if I am here, at least a Nodo did not answer in time

// find which Nodo did not answer
String escludi_nodo=trova_nodo_mancante();

  // memorize that the original path is blocked
  percorso_originale_interrotto_=true;

  // if I have already contacted the PA, I have a list with all the paths
  // and I was trying to reserve path number num_percorso_: so let's
  // memorize that this path is blocked (total delay = -1) invoking p.setRitardo(-1)
  if(flag_percorso_alternativo_){
      Percorso_alternativo p=(Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
      p.setRitardo(-1);
  }

  ritardo_accumulato_ = max_ritardo_acc_ + 1;

// reset the lists and the variables I use to control the reservation process
  passi_confermati_ = 0;
  passi_risposti_ = 0;
  lista_prenotazioni_accettate_ = new Vector();
  lista_prenotazioni_non_accettate_ = new Vector();
  lista_prenotazioni_richieste_ = new Vector();
  lista_prenotazioni_confermate_ = new Vector();

  // update my position on the graph and let's search an alternative path
  aggiorna_posizione();
  // avoid the Nodo that did not answer
  cerca_percorso_alternativo(escludi_nodo);
}
```

Later on we report the code of method *cerca_percorso_alternativo(String avoid_node)*.

```
private void cerca_percorso_alternativo(String escludi_nodo){
  // parametri:
  // mio nome; nodo iniz; nodo fin; nodo sosta; [nodo1, nodo2...] percorso corrente
  // devo inviare la richiesta a percorsi alternativi se e solo se non l'ho
  // gia' inviata in questa ricerca di percorso

// if I have not yet asked the PA, let's send a request
  if(inviata_richiesta_PA_ == false){
    inviata_richiesta_PA_ = true;
    lista_prenotazioni_accettate_ = new Vector();
```

```
    lista_prenotazioni_non_accettate_ = new Vector();
    lista_prenotazioni_richieste_ = new Vector();
    String percorso = "[";
    String nodo_sosta = "";
    String ultimo_nodo = "";
    String primo_nodo = "";

    // find the node where I am
    // (this list has been updated by the aggiorna_posizione())
    // so the first element of the list is where I really am
    PassoTreno app = (PassoTreno)lista_passi_corrente_.elementAt(0);
    primo_nodo = app.getNodo();
    percorso = percorso + primo_nodo + ",";

    // in String "percorso" I prepare the list of my current path
    // and at the same time I search the Stop node, if any
    for(int i = 1;i < lista_passi_corrente_.size();i++){
      app = (PassoTreno)lista_passi_corrente_.elementAt(i);
      if(app.isSosta()){
        // stop node
        nodo_sosta = app.getNodo();
      }
      percorso = percorso + app.getNodo() + ",";
    }
    ultimo_nodo = app.getNodo();
    if(ultimo_nodo.equals(nodo_sosta)){
      nodo_sosta = "";
    }
    if(primo_nodo.equals(nodo_sosta)){
      nodo_sosta = "";
    }
    percorso = percorso.substring(0,percorso.length() - 1) + "]";
    ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
    msg.addReceiver(new AID("PercorsiAlternativi",false));
    msg.setSender(this.getAID());

    // add the other information for PA and send the message
    msg.setContent(nomeTreno_ + ";" + primo_nodo + ";" + ultimo_nodo +
            ";" + nodo_sosta + ";" + percorso + ";" + escludi_nodo);
    send(msg);

  }else{
   // if I have already contacted PA I will only choose the next alternative path
    lista_prenotazioni_accettate_ = new Vector();
    lista_prenotazioni_non_accettate_ = new Vector();
    lista_prenotazioni_richieste_ = new Vector();

    if(lista_percorsi_alternativi_.size() <= (num_percorso_)){
      // there are no more alternative path...
      gestisci_no_percorsi_alternativi();

    }else{
// this class represents an alternative path.
// let's take the Vector representing the list of nodes
// and let's try to reserve this new path
      Percorso_alternativo nuovo_percorso =
              (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_);
      prenota_percorso(nuovo_percorso.getLista_passi_());
      num_percorso_++;
    }
  }
}
```

```
private void gestisci_no_percorsi_alternativi(){
    long max_ritardo_attuale =100000000;
    int percorso_migliore=-1;

    // search the list of alternative path to find the one with the lowest delay

    for(int i=0;i<lista_percorsi_alternativi_.size();i++){
          Percorso_alternativo percorso =
           (Percorso_alternativo)lista_percorsi_alternativi_.elementAt(i);

          if(percorso.getRitardo()>-1 && percorso.getRitardo()<max_ritardo_attuale){
              max_ritardo_attuale=percorso.getRitardo();
              percorso_migliore=i;
          }
      }


    if(percorso_migliore==-1){
      // I cannot find a path with an acceptable delay
      sono_bloccato=true;
inviata_richiesta_PA_ = true;

      num_percorso_=1;    // restart from the first alternative path

// reset the delay of all the path
      for(int i=0;i<lista_percorsi_alternativi_.size();i++){
          Percorso_alternativo percorso =
           (Percorso_alternativo)lista_percorsi_alternativi_.elementAt(i);
          percorso.setRitardo(-1);
      }

      passi_confermati_ = 0;
      passi_risposti_ = 0;

      // consider again the path I was considering before asking PA
      lista_passi_corrente_=lista_passi_prima_di_PA;

    if(percorso_originale_interrotto_){

// if the previous path was blocked:

// update my position on the graph
        aggiorna_posizione();

// send a reservation request NON disputable to the Node where I am
// to say that I must stay there fo more than planned
          PassoTreno passoCorrente=(PassoTreno)lista_passi_corrente_.elementAt(0);
          ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
          msg.addReceiver(new AID(passoCorrente.getNodo(),false));
          msg.setSender(this.getAID());

          prgPrenotazioni_++;
          // I reserve the resource for more than planned, adding a default time delta_stop_
          // to the previous departure time. This because if this path is blocked, is unusefull
          // keep trying on reserving it now. Let's wait and try again later...
          msg.setContent(prgPrenotazioni_ + ";" + passoCorrente.getOraPartenzaNodoPrecedente()
                + ";" + (passoCorrente.getOraPartenzaNodo() + delta_stop_) + ";" +
                passoCorrente.getNodo() + ";false;" + priorita_);
          send(msg);

          // deleta the previous other reservations
          lista_prenotazioni_richieste_ = new Vector();
```

```
          // add the reservation requet to the list
          inserisci_prenotazione(prgPrenotazioni_,
                                 passoCorrente.getNodo(),
                                 0,   // itinerario
                                 passoCorrente.getOraPartenzaNodoPrecedente(),
                                 passoCorrente.getTipoOraPartenzaNodoPrecedente(),
                                 passoCorrente.getOraArrivoNodo(),
                                 passoCorrente.getTipoOraArrivoNodo(),
                                 passoCorrente.getOraPartenzaNodo() + delta_stop_,
                                 passoCorrente.getTipoOraPartenzaNodo(),
                                 false,
                                 false,
                                 lista_prenotazioni_richieste_);

// let's shift all the current path of delta_stop_ time
// so when I will start again to reserve the arrival and departure time
// will be correct
          ritarda_percorso(lista_passi_corrente_, delta_stop_);

// reset the variable
          flag_percorso_alternativo_=true;
          flag_percorso_alternativo_confermato_=false;
          flag_percorso_alternativo_suggerito_=false;
          percorso_originale_interrotto_=false;
          cerco_percorso_ = true;
          passi_confermati_ = 0;
          passi_risposti_ = 0;
          ritardo_accumulato_ = 0;
          sono_bloccato=true;
          accetta_comunque_ritardo=false;
          lista_prenotazioni_confermate_ = new Vector();
          lista_prenotazioni_accettate_ = new Vector();
          lista_prenotazioni_non_accettate_ = new Vector();
          lista_passi_aggiornata_ = new Vector();
          inviata_richiesta_PA_ = true;
          num_percorso_ = 1;
          treno_concorrente_="";

          // suspend myself and wait
          controlla_bloccato_ = new WakerBehavior(this, 5000) {
            protected void onWake(){
// when I wake up, let's start again trying to reserve the path
              aggiorna_posizione();

              cerco_percorso_ = true;

              PassoTreno passoCorrente=(PassoTreno)lista_passi_corrente_.elementAt(0);

              // try to reserve the path, with out the first node, because I'm already on it
              prenota_percorso((Vector)lista_passi_corrente_.subList(1, lista_passi_corrente_.size()));
              passi_confermati_ = 1;
              passi_risposti_ = 1;

              lista_passi_corrente_.add(0, passoCorrente);

            }
          };
          addBehavior(controlla_bloccato_);

      }else{
```

```
// if the previous path was not blocked:
// accept more delay
          accetta_comunque_ritardo=true;
          flag_percorso_alternativo_=false;

         // delay the previous path (before asking PA)
          ritarda_percorso(lista_passi_prima_di_PA, ritardo_accumulato_);

// let's wait on the first node a little more
          ((PassoTreno)lista_passi_prima_di_PA.get(0)).
           setOraPartenzaNodoPrecedente(currentTimeMillis() - deltaSincronizzazione_);

          // try to reserve this new path (different arrival and departure times)
          prenota_percorso(lista_passi_prima_di_PA);
      }
    }
    else{
// I'm not blocked: I choose the alternative path with the minimal delay
        sono_bloccato=false;
        num_percorso_=percorso_migliore;

        // I'm managing a situation where all paths have a huge delay: the one that
        // I'm going to try to reserve will have again a huge delay --> memorize that I must accept it
        accetta_comunque_ritardo=true;

        // take the best path and send the reservation requests
        prenota_percorso(((Percorso_alternativo)lista_percorsi_alternativi_.
         get(percorso_migliore)).getLista_passi_());
    }
}
```

Later on we report the code of method $gestisci\_aggiornamenti\_nodi(msg)$.

We start with $gestisci\_INFORM\_da\_nodo$.

```
private void gestisci_aggiornamenti_nodi(ACLMessage msg){

// let's call a different method with respect to the performative
  int perf = msg.getPerformative();
  if(perf == 7){
    gestisci_INFORM_da_nodo(msg);
  }else if(perf == 4){
    gestisci_CONFIRM_da_nodo(msg);
  }else if(perf == 2){
    gestisci_CANCEL_da_nodo(msg);
  }else if(perf == 6){
    gestisci_FAILURE_da_nodo(msg);
  }else if(perf == 16){
    gestisci_REQUEST_da_nodo(msg);
  }else if(perf==21){
    gestisci_PROPAGATE_da_nodo(msg);
    // it simply invoke a method that find the reservation and
    // change the arc to be used
  }
}
```

```
private void gestisci_INFORM_da_nodo(ACLMessage msg){

    String nodo = msg.getSender().getLocalName();
    String con = msg.getContent();
    int x = con.indexOf(";");
    int numPrenotazione = Integer.parseInt(con.substring(0,x));
    String ncon = con.substring(x + 1,con.length());
    x = ncon.indexOf(";");
    boolean accettata = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");

    // here there are the new arrival and departure time when the node will be free
    long oraInizioDisponibilitaNodo = Long.parseLong(ncon.substring(0,x));
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");
    long oraFineDisponibilitaNodo = Long.parseLong(ncon.substring(0,x));
    ncon = ncon.substring(x + 1,ncon.length());

    x = ncon.indexOf(";");
    int priorita = Integer.parseInt(ncon.substring(0,x));
    ncon = ncon.substring(x + 1,ncon.length());
    x = ncon.indexOf(";");
    String altro_treno = ncon.substring(0,x);
    ncon = ncon.substring(x + 1,ncon.length());
    int it = Integer.parseInt(ncon);

// check if this reservation request is still valid (no time-out elapsed).
// if the methods returns -1 the request is old --> return
    long oraFinePrenotazione = recupera_a(numPrenotazione);
    if(oraFinePrenotazione == -1){
      return;
    }

    String escludi_nodo=msg.getSender().getLocalName();

// update the number of total answer from Nodo agents
    passi_risposti_++;
    if(priorita < this.priorita_ || altro_treno.equals(treno_concorrente_)){
      // If my priority is lower than the one of the other Train, I cannot steal

// if the Nodo specified a new free interval
      if(oraInizioDisponibilitaNodo != 0 && oraInizioDisponibilitaNodo != -1){

        // move this message in those not accepted
        rifiuta_richiesta(numPrenotazione,
                          oraInizioDisponibilitaNodo,
                          oraFineDisponibilitaNodo);

        // calculate the current delay
        ritardo_accumulato_ = ritardo_accumulato_ +
          (oraFineDisponibilitaNodo - oraFinePrenotazione);

        if(ritardo_accumulato_ > max_ritardo_acc_ && accetta_comunque_ritardo==false){
          // if the delay is too high and I must not accept it anyway,
          // let's stop waiting and let's find another solution
          Thread th = getThread(controlla_timer_);
          th.interrupt();

          // if it was already an alternative path, memorize the total delay,
```

```
    if(flag_percorso_alternativo_){
       Percorso_alternativo p=
        (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
       p.setRitardo(ritardo_accumulato_);
    }
    else percorso_originale_interrotto_=false;

    aggiorna_posizione();

    // ask PA for other alternative path or take the next alternative path from the list
    cerca_percorso_alternativo(escludi_nodo);

  }else{

    // the total delay is still acceptable

    if(passi_risposti_ == lista_passi_corrente_.size()){
      // if I have already received all the answers

      // stop the thread because I have all the answer, no more need to wait
      Thread th = getThread(controlla_timer_);
      th.interrupt();

      // update the total delay of the current alternative path
      if(flag_percorso_alternativo_){
         Percorso_alternativo p=
          (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
         p.setRitardo(ritardo_accumulato_);
      }

      // update the reservation request with respect to the delay
      // and try to reserve this new parh.
      // It invokes the prenota_percorso(path)
      ricalcola_percorso();

    }else{
      // I need to wait more, for all the answers to arrive

    }
  }
}else{
  // If the node does not specify a new free interval

  Thread th = getThread(controlla_timer_);
  th.interrupt();

  // move this message in those not accepted
  rifiuta_richiesta(numPrenotazione,
                    oraInizioDisponibilitaNodo,
                    oraFineDisponibilitaNodo);

  // in this case the total dely of this path will be -1,
  // tath is, too much delay
  if(flag_percorso_alternativo_){
      Percorso_alternativo p=
       (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
      p.setRitardo(-1);
  }
  // otherwise, the original path is blocked
  else percorso_originale_interrotto_=true;

  aggiorna_posizione();
```

```
        // ask PA for other alternative path or take the next alternative path from the list
        cerca_percorso_alternativo(escludi_nodo);
      }

    }else if(priorita >= this.priorita_ && altro_treno.equals(treno_concorrente_)!=true){
      // My priority is higher than the one of the train that has the resource

// if the total delay is too high --> steal, otherwise, shift the reservation (do not steal)
      if(oraInizioDisponibilitaNodo != 0 && oraInizioDisponibilitaNodo != -1){

        ritardo_accumulato_ = ritardo_accumulato_ +
          (oraFineDisponibilitaNodo - oraFinePrenotazione);

        if(ritardo_accumulato_ > max_ritardo_acc_ && accetta_comunque_ritardo==false){

            // too delay, steal!!
            // memorize the train I'm stealing a resource from
            treno_concorrente_=altro_treno;

            passi_confermati_++;
            if(passi_risposti_ == lista_passi_corrente_.size()){
              // I have all the answers
              Thread th = getThread(controlla_timer_);
              th.interrupt();

              if(flag_percorso_alternativo_){
                  Percorso_alternativo p=
                   (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
                  p.setRitardo(ritardo_accumulato_);
              }

              if(passi_confermati_ == passi_risposti_){
               // all the answer are "ok"
// send a confirm message to all nodes in the path
                conferma_percorso();
                cerco_percorso_ = false;
                percorso_originale_interrotto_=false;
                ritardo_accumulato_ = 0;
              }else{
               // some of the answers are not "ok"
                cerco_percorso_ = true;
                // update the arrival/departure time of the path with respect
                // to the reservation requests that cannot be accepted
                ricalcola_percorso();
              }
            }else{
             // Not all the answers are arrived
              cerco_percorso_ = true;
            }
          }
        else{

         // the total delay is acceptable

            if(passi_risposti_ == lista_passi_corrente_.size()){

              Thread th = getThread(controlla_timer_);
              th.interrupt();

            if(flag_percorso_alternativo_){
               Percorso_alternativo p=
```

```
                (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
              p.setRitardo(ritardo_accumulato_);
            }

              // update the arrival/departure time for all the nodes in the path
              ricalcola_percorso();

            }else{
              // I must wait for all the answers

              // memorize that the request cannot be accepted
              rifiuta_richiesta(numPrenotazione,
                        oraInizioDisponibilitaNodo,
                        oraFineDisponibilitaNodo);
            }
          }
      }
      else{
          // I cannot steal and the node did not indicate a new interval

          Thread th = getThread(controlla_timer_);
          th.interrupt();
          if(flag_percorso_alternativo_){
              Percorso_alternativo p=
               (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
              p.setRitardo(-1);
          }
          else percorso_originale_interrotto_=true;

          aggiorna_posizione();

          // search for an alternative path or select the next one
          cerca_percorso_alternativo(escludi_nodo);

      }
    }
}
```

Later on we report the code of method $gestisci\_CONFIRM\_da\_nodo$ and of the method $gestisci\_CANCEL\_da\_nodo$.

```
private void gestisci_CONFIRM_da_nodo(ACLMessage msg){

    String nodo = msg.getSender().getLocalName();
    String con = msg.getContent();
    int x = con.indexOf(";");
    int numPrenotazione = Integer.parseInt(con.substring(0,x));
    String ncon = con.substring(x + 1,con.length());
    x = ncon.indexOf(";");
    boolean accettata = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
    ncon = ncon.substring(x + 1,ncon.length());
    int it = Integer.parseInt(ncon);

// if my reservation request is alredy expired then exit
    boolean trovata = accetta_richiesta(numPrenotazione,it,false);
    if(trovata == false){
      return;
    }
```

```
    // increment the number of responses and that of the confirmed requests
    passi_confermati_++;
    passi_risposti_++;

      if(passi_risposti_ == lista_passi_corrente_.size()){
      // all the Nodo answered

      Thread th = tbf_.getThread(controlla_timer_);
      th.interrupt();

      if(passi_confermati_ == passi_risposti_){ // tutte le risposte sono positive
        // all the answers are CONFIRM
        conferma_percorso();
        flag_percorso_alternativo_confermato_=true;
        percorso_originale_interrotto_=false;

        // I'm ok, I have a confirmed path --> if no Nodo will cancel my reservation,
        // I have no more duties to do
        cerco_percorso_ = false;
        ritardo_accumulato_ = 0;

        dove_sono_=0;

      }else{
        // not all the answers are CONFIRM (not all the resources are available)

        // I cannot confirm the path
        cerco_percorso_ = true;

        // update the arrival and departure time and try again to reserve the path
        ricalcola_percorso();

      }
    }else{
     // not all the answers are arrived, I must wait
      cerco_percorso_ = true;
    }
  }



private void gestisci_CANCEL_da_nodo(ACLMessage msg){

  String con = msg.getContent();
  int x = con.indexOf(";");
  int numPrenotazione = Integer.parseInt(con.substring(0,x));
  String ncon = con.substring(x + 1,con.length());
  x = ncon.indexOf(";");
  boolean accettata = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  long oraInizioDisponibilitaNodo = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  long oraFineDisponibilitaNodo = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  int pri = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  String altro_tr = ncon;
```

```
  aggiorna_posizione();

// update my status tomemorize that I am searching for a path
  cerco_percorso_ = true;

  String escludi_nodo=msg.getSender().getLocalName();

// move the confirmed reseravation into the list of those not accepted
  annulla_prenotazione(numPrenotazione,oraInizioDisponibilitaNodo,oraFineDisponibilitaNodo);

  // make the confirmed reservations only accepted
  lista_prenotazioni_accettate_ = lista_prenotazioni_confermate_;

  // send a cancel message to the remaining nodes
  annulla_percorso(lista_prenotazioni_confermate_);

  lista_prenotazioni_confermate_ = new Vector();

if(oraInizioDisponibilitaNodo != 0 && oraInizioDisponibilitaNodo != -1){
   // if the Nodo specified also a new interval I can recalculate the time intervals

    ritardo_accumulato_ = ritardo_accumulato_ + (oraFineDisponibilitaNodo - va);
    if(ritardo_accumulato_ > max_ritardo_acc_ && accetta_comunque_ritardo==false){
// if the delay it too much let's ask a new path (or a new from the list)

      if(flag_percorso_alternativo_){
          Percorso_alternativo p=
           (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
          p.setRitardo(-1);
      }

      cerca_percorso_alternativo(escludi_nodo);

    }else{
  // the total delay is acceptable

      if(flag_percorso_alternativo_){
          Percorso_alternativo p=
           (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
          p.setRitardo(ritardo_accumulato_);
      }

// shift the reservation for this path and send again the reservation requests
      ricalcola_percorso();
    }
  }else{

// if the Nodo did not specifed a new free interval
    if(flag_percorso_alternativo_){
         Percorso_alternativo p=
          (Percorso_alternativo)lista_percorsi_alternativi_.get(num_percorso_ - 1);
         p.setRitardo(-1);
     }

     // I'm blocked and I must search/use an alternative path
     else percorso_originale_interrotto_=true;
     cerca_percorso_alternativo(escludi_nodo);
  }
}
```

We conclude with the code of method $gestisci\_FAILURE\_da\_nodo$

```
private void gestisci_FAILURE_da_nodo(ACLMessage msg){
// This message means that I'm on a node that has gone out of order

  String con = msg.getContent();
  int x = con.indexOf(";");
  int numPrenotazione = Integer.parseInt(con.substring(0,x));
  String ncon = con.substring(x + 1,con.length());
  x = ncon.indexOf(";");
  boolean accettata = Boolean.valueOf(ncon.substring(0,x)).booleanValue();
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  long oraInizioDisponibilitaNodo = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  long oraFineDisponibilitaNodo = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  x = ncon.indexOf(";");
  int pri = Integer.parseInt(ncon.substring(0,x));
  ncon = ncon.substring(x + 1,ncon.length());
  String altro_tr = ncon;

  long va = recupera_a_conf(numPrenotazione);
  if(va < oraFineDisponibilitaNodo){

   // the out of order status ends after my departure time -->
   // I must wait on the node!

      // reset my situation
      sono_bloccato = true;
      cerco_percorso_= true;
      long delta=oraFineDisponibilitaNodo - va;

// I must update my confirmed reservation for this node and update the
// departure time (I will leave at oraFineDisponibilitaNodo)
      PassoTreno aggiornato=(PassoTreno)lista_passi_corrente_.elementAt(0);
      aggiornato.setOraPartenzaNodo(oraFineDisponibilitaNodo);
      lista_passi_corrente_.set(0,aggiornato);

// then I must cancel the remaining path
// and try to reserve it again but with the new arrival/departure times.
// Use some temporary lists (resto_di_lista_passi_corrente and resto_del_percorso)

      Vector resto_del_percorso= new Vector();
      Vector resto_di_lista_passi_corrente= new Vector();
      Prenotazione_Treno pr_agg= new Prenotazione_Treno();
      PassoTreno passo_agg= new PassoTreno();

      for(int n = 0;n < lista_prenotazioni_confermate_.size();n++){
          if(((Prenotazione_Treno)lista_prenotazioni_confermate_.elementAt(n))
           .getNodo().equals(dove_sono)){

              pr_agg=((Prenotazione_Treno)lista_prenotazioni_confermate_.get(n));
              pr_agg.setOraPartenzaNodo(oraFineDisponibilitaNodo);
              passo_agg=(PassoTreno)lista_passi_corrente_.get(n);
              passo_agg.setOraPartenzaNodo(oraFineDisponibilitaNodo);
              lista_passi_corrente_.set(0, (PassoTreno)passo_agg);
          }
```

```
          else{
              resto_del_percorso.add((Prenotazione_Treno)lista_prenotazioni_confermate_.elementAt(n));
              resto_di_lista_passi_corrente.add((PassoTreno)lista_passi_corrente_.get(n));
          }
      }

      // cancel the other confirmed reservations
      annulla_percorso(resto_del_percorso);

      lista_prenotazioni_confermate_ = new Vector();
      lista_prenotazioni_confermate_.add(pr_agg);

// shift the arrival/departure time
      ritarda_percorso(resto_di_lista_passi_corrente, delta);

      // try to reserve this new path
      prenota_percorso(resto_di_lista_passi_corrente);

      passi_confermati_ = 1;
      passi_risposti_ = 1;

// update the first step of the path (where I am)
      lista_passi_corrente_.add(0, passo_agg);

  }
  else{
   // if the out of order status ends before I leave, it does not affect anything
  }

}
```

Appendix C

Simplified code of the User Agents Manager

```
// global variable
deltaSincronizzazione_=0;

void setup(){

    Delta= 180000;

// this is my Zero Time, that is the time when the simulation is started,
// that must be common to all trains (and resource agents)
deltaSincronizzazione_=(currentTimeMillis());


    get_train_list();

// check if there are already trains to be created
    create_trains();

    controlla_timer_ = new TickerBehavior((Delta/2)) {
      // create a Behavior that will check, every Delta/2,
      // if there are train to be created
     void onTick(){
        create_trains();
      }
    };

    addBehavior(controlla_timer_));
  }

  void get_train_list(){

// get the connection to the database
        connect_to_database();

        ResultSet srs = executeQuery(
        SELECT treni.id_treno, pr, ritardo, direzione, ora_arrivo
        FROM public.treni inner join public.tabelle_marcia
```

```
        on treni.id_treno = tabelle_marcia.id_treno
        WHERE tappa = 1 order by (ritardo + ora_arrivo));

        elenco_treni= new Array();

        while (srs.next()) {

// Structure with the information regarding the Train just read from the db.
            Dati mio_treno = new Dati(srs);
            // in mio_treno there is a field named OraTrenoOrigine
            // that will be inizialized with the value   srs.getLong(5) + srs.getLong(3),
            // that is the real time when the train will appear out of the graph.
            // The path will be read from the database only when the Agent will be created.

            elenco_treni.add(mio_treno);
        }
}
```

```
void create_trains(){

// class Dati is used to keep the information that a Train agent will need
    Dati treno;

    for(int n = 0; n < elenco_treni.size() ; n++){
        treno= (Dati)elenco_treni.get(n);

        // if the train will enter the graph within Delta let's create it
        if(treno.OraTrenoOrigine < (currentTimeMillis()-deltaSincronizzazione + Delta) ){
            elenco_treni.remove(n);
            n--;

// parameters for the agent to be created
            Object args[] = new Object[2];

// read from the database the original path of the agent

   connect_to_database();

            ResultSet srs = executeQuery(
            SELECT id_nodo, ora_arrivo, ora_partenza, ora_arrivo_n, ora_partenza_n,
            tappa, sosta, entrata, uscita
            FROM tabelle_marcia inner join nodi on nodi.id=tabelle_marcia.id_nodo WHERE
            tabelle_marcia.id_treno= " + treno.IdTreno + " order by tappa);

            List tappe= new Array();

            while (srs.next()) {

                // when this object is created it initializes its fields with
                //the value read from the database
                Nodo mia_tappa= new Nodo(srs);

                // note that the delay must be added to the original arrival and departure times
                mia_tappa.setOraArrivo(srs.getLong(2) + treno.Delay);
                mia_tappa.setOraPartenza(srs.getInt(3) + treno.Delay);

                tappe.add(mia_tappa);
            }

// the field TDM is a list where the train has its original plan
            treno.setTDM(tappe);

// the first parameter is the object with all the information of the train
            args[0] = treno ; //dett;

            // the second parameter is the Zero Time of the simulation
            args[1]= deltaSincronizzazione_;

// JADE like code, it will set up a new JADE Agent
            Agent dummy=createNewAgent("Treno_" + treno.IdTreno, args);

            // Fire up the agent
            dummy.start();

        }// end while

    }// end if

}
```

Appendix D

Simplified code of the Resource Agents Manager

```
private Vector lista_mess;
private Vector lista_act;
private incompatibilita ListaIncompatibilita = new incompatibilita();
private ArrayList listaNodi;
private long miozero_;
private ArrayList elenco_rotture_;
private WakerBehavior crea_rottura_;

protected void setup(){
  // this is the Zero time
  miozero_=System.currentTimeMillis();

  lista_mess = new Vector();
  lista_act = new Vector();

  carica_elenco_rotture();
  serveInizializzaStazioneAction();
}


private void carica_elenco_rotture(){
      connect_to_database();

      ResultSet srs = executeQuery(
      "SELECT id_nodo, da, a, rotto_anche_nodo, itinerario, da_nodo " +
      " FROM rotture_nodi order by da");

      elenco_rotture_= new ArrayList();

      // save in this list all the rows from database
      while (srs.next()) {
          // this class represents a row from the table "rotture_nodi"
          Rottura r;
          r= new Rottura(srs.getInt(1),
                  srs.getLong(2),
                  srs.getLong(3),
                  srs.getBoolean(4),
```

```
                    srs.getInt(5),
                    srs.getString(6));

            elenco_rotture_.add(r);
        }

        if(elenco_rotture_.size()>0){
            Iterator it = elenco_rotture_.iterator();

// calculate, with respect to the Zero time, when the first "out of order status" will start
            long prossima_rottura=((Rottura)elenco_rotture_.get(0)).getDa() -
            (System.currentTimeMillis() - miozero_);

// add a WakerBehavior that will start at the next foreseen "out of order status"
// and that will send the notification to the Nodo agent
            crea_rottura_ = new WakerBehavior(this, prossima_rottura){
              protected void onWake(){ //handleElapsedTimeout(){
                     // call the method to send the notification
                     crea_rottura();
              }
            };
            addBehavior(crea_rottura_);
        }
}

private void crea_rottura(){
      Rottura r; Failure f;
      long t=System.currentTimeMillis() - miozero_;
      boolean it_rotto=false;

      for(int i=0;i<elenco_rotture_.size();i++){
          r=(Rottura)elenco_rotture_.get(i);

          // t + 2000 because we must consider the execution time,
          // so we can not check the exact value r.getDa()==t
          if(r.getDa()<= t && r.getDa()<= t + 2000){

              if(r.getItinerario()==-1) it_rotto=false;
              else it_rotto=true;

// class Failure keep all the information concerning a "out of order status"
              f=new Failure();
              f.setNomeNodoFailure("Nodo_" + r.getId_nodo());
              f.setIdItinerarioFailure(r.getItinerario());
              f.setNomeNodoProvenienza(r.getDa_nodo());
              f.setFlagNodoFailure(r.isRotto_anche_nodo());
              f.setFlagItinerarioFailure(it_rotto);
              f.setTimeInizio(r.getDa());
              f.setTimeFine(r.getA());

            // send message to the node
            ACLMessage msg2=new ACLMessage(ACLMessage.INFORM);

            // format of the content
            // $data;from;to;it;fromNodo;NodoOut$
            msg2.setContent(System.currentTimeMillis() +";"+
            fail.getTimeInizio() +";"+fail.getTimeFine() +";"+
            fail.getIdItinerarioFailure() + ";" + fail.getNomeNodoProvenienza() +";" +
            fail.getFlagNodoFailure());

            msg2.addReceiver(new AID(fail.getNomeNodoFailure(),false));
            send(msg2);
```

```
            // delete this item from the list
            elenco_rotture_.remove(i);
          }
          else break;
      }

// create the new WakerBehavior for the future failures
      if(elenco_rotture_.size()>0){
          long prossima_rottura=((Rottura)elenco_rotture_.get(0)).getDa() - t;

          crea_rottura_ = new WakerBehavior(this, prossima_rottura){
          protected void onWake(){
              crea_rottura();
          }};
        addBehavior(crea_rottura_);
      }
  }


private void serveInizializzaStazioneAction()
  {
// this method reads from the database the list of Nodes
// and inserts them in list listaNodi
    leggi_lista_nodi();

// this method reads from the database the list of incompatibilities
// and insert them in list listaItinerari
    leggi_lista_incompatibilita();

    while(listaNodi.hasNext()){
      Object args[] = new Object[3];

      // nodo class will keep the basic information on a Nodo
      nodo dett = (nodo)listaNodi.next();
      String nomeNodo = "Nodo_" + dett.getNomeNodo();
      args[0] = dett;

// Here in this simplified code we do not report the complete management
// of arcs and incompatibilities because the detail level should be too low and not interesting.
      args[1] = listaItinerari;

// this will be the Zero Time
      args[2] = miozero_ ;

// JADE like code, it will set up a new JADE Agent
      Agent dummy=createNewAgent(nomeNodo, args);

      // Fire up the agent
      dummy.start();
     }
  }
```

Appendix E

Simplified code of the Paths Manager

```
  ACLMessage msg;

  // ordered list with all the path
  private List elenco_percorsi_;

  // working list, with the reversed paths (to be added to the final list)
  private List elenco_percorsi_2_;

  // list of Stop "Nodes"
  private List nodi_sosta_;

protected void setup(){
// read from database the data and organize them
    caricaPercorsi();

    // add the CyclicBehavior that wait for Trains requests for alternative paths
    addBehavior(new CyclicBehavior(this) {
      public void action(){

        msg = myAgent.receive();

// format of the content
        // PriorityOfTrain;StartingNode;EndingNode;StopNode;
        // CurrentPath ([Node1, Node2...]);SkipNode
// PriorityOfTrain is no more used in this version of the system

// parse the string
          String pri, primo, secondo, sosta, lista, escludi_nodo;
          int x;
          String con = msg.getContent();
          x = con.indexOf(";");
          pri = con.substring(0,x);
          con = con.substring(x + 1,con.length());
          x = con.indexOf(";");
          primo = con.substring(0,x);
          con = con.substring(x + 1,con.length());
          x = con.indexOf(";");
          secondo = con.substring(0,x);
          con = con.substring(x + 1,con.length());
          x = con.indexOf(";");
```

```
          sosta = con.substring(0,x);
          con = con.substring(x + 1,con.length());
          x = con.indexOf(";");
          lista = con.substring(0,x);
          con = con.substring(x + 1,con.length());
          escludi_nodo = con.substring(0,con.length());

          // call the method that finds the alternative paths and sends back them to the train
          (primo,secondo,sosta,pri,lista,escludi_nodo);
        }
      }
    });
  }




private void caricaPercorsi(){

connect_to_database();

        ResultSet srs = executeQuery(
         "SELECT id FROM nodi where sosta= true order by id");

        nodi_sosta_= new ArrayList();
        while (srs.next()) {

            nodi_sosta_.add(String.valueOf(srs.getInt(1)));

        }

        srs.close();

        srs = executeQuery("SELECT id FROM nodi order by id");

        // 2 matrices
        elenco_percorsi_= new ArrayList();
        elenco_percorsi_2_= new ArrayList();

        while (srs.next()) {

            List percorso= new ArrayList();
            percorso.add(String.valueOf(srs.getInt(1)));
            dfs(srs.getInt(1), percorso);

        }

// this method moves every element of the list elenco_percorsi_2_
// into the final list elenco_percorsi_,
// ordering all paths with respect to the name of the first node of the path
        merge_percorsi();

  }


  private void dfs(int id_nodo, List percorso){

        ResultSet srs = executeQuery(
         "SELECT distinct id_nodo_uscente, id_nodo_entrante, entrata, uscita " +
          " FROM itinerari inner join nodi on nodi.id=itinerari.id_nodo_entrante " +
```

```
          " WHERE id_nodo_uscente= " + id_nodo + " order by id_nodo_entrante");

        if(percorso.size()>1){
            elenco_percorsi_.add(percorso);

// note that this code is correct if, and only if,
// we assume, as in "Stand-alone FYPA", that all arcs are bidirectional
            List percorso_rev= new ArrayList();
            for(int i= percorso.size();i>0;i--){
                percorso_rev.add(percorso.get(i-1));
            }

            elenco_percorsi_2_.add(percorso_rev);

        }




      while (srs.next()){

            List npercorso= new ArrayList();
            for(int i= 0;i<percorso.size();i++){
                npercorso.add(i, (String)percorso.get(i));
            }

            npercorso.add(String.valueOf(srs.getInt(2)));

            // note that this code is correct only if we inserted in the database
            // arcs all in one traveling direction!

            dfs(srs.getInt(2), npercorso);

        }

  }
```

Subsequently we report the simplified code of method $recuperaPercorsi$, that finds the alternative paths and sends back them to the train.

```
private void recuperaPercorsi(
String primo,String secondo,String sosta,String lista, String escludi_nodo){

  // parse the parameter lista that contains the current path of train
  // (to be excluded from the returned list)

    String con = lista;
    String nodo;
    Vector lista_nodi = new Vector();
    if(con.charAt(1) != ']'){
      int x = 0;
      con = con.substring(1,con.length());
      while(true){
        x = con.indexOf(",");
        if(x == -1){
          nodo = con.substring(0,con.length() - 1);
          lista_nodi.add(nodo);
          break;
        }else{
          nodo = con.substring(0,x);
          lista_nodi.add(nodo);
          con = con.substring(x + 1,con.length());
        }
      }
    }

      List ret=trova_percorsi(primo, secondo, sosta, lista_nodi, escludi_nodo);

      for(i = 0;i < ret.size();i++){
        txt = txt + "(";
        // NodoCnf is a class that represents a Node with all the information needed
        NodoCnf[] listaNodi = ((Percorso)ret.get(i)).getListaNodi();
        for(int j = 0;j < listaNodi.length;j++){
          txt=txt+listaNodi[j].getNomeNodoCnf()+"_"+listaNodi[j].isFlagSosta()+";";
        }
        txt = txt.substring(0,txt.length() - 1) + "),";
      }

      if(ret.size()>0) txt = txt.substring(0,txt.length() - 1) + "]";
      else txt = txt + "]";

      // msg is a global variable that contains the received message from the train
      ACLMessage rep = msg.createReply();
      rep.setPerformative(ACLMessage.INFORM);
      rep.setContent(txt);
      this.send(rep);
  }

// this method looks for the subset of paths conforming to the parameters
// and return them
  private List trova_percorsi(
  String primo, String ultimo, String sosta, Vector lista_nodi, String escludi_nodo){

      List risultati = new ArrayList();
      List percorso= new ArrayList();
      boolean trovato_nodo_escluso=false;
```

```
      for(int i=0; i<elenco_percorsi_.size();i++){
          trovato_nodo_escluso=false;
          percorso=(List)elenco_percorsi_.get(i);

          // check if the first and the last node of this path are equal to those requested
          if(("Nodo_" + (String)percorso.get(0)).equals(primo) &&
          ("Nodo_" + (String)percorso.get(percorso.size()-1)).equals(ultimo)){

// this class represents a path: has an identification number (a counter)
// and a list of NodoCnf objects representing the nodes of the path
              Percorso nuovo= new Percorso();
              nuovo.setIdPercorso(i);
              NodoCnf[] passi= new NodoCnf[percorso.size()];
              for(int j=0; j<percorso.size();j++){
                  NodoCnf nodo= new NodoCnf();
                  nodo.setIdNodoCnf(Integer.valueOf((String)percorso.get(j)));
                  nodo.setNomeNodoCnf((String)percorso.get(j));
                  nodo.setFlagSosta(sosta((String)percorso.get(j)));
                  passi[j]=nodo;

// check if the node is equal to the one that the train needs to avoid
                  if(("Nodo_" + (String)percorso.get(j)).equals(escludi_nodo))
                   {trovato_nodo_escluso=true;}

                }

              nuovo.setListaNodi(passi);

              if(trovato_nodo_escluso==false){

                  // check if this path is equal to the current one of the train
                  for(int x=0; x<lista_nodi.size();x++){
                      if(((String)lista_nodi.get(x)).equals
                      ("Nodo_" + String.valueOf(((NodoCnf)passi[x]).getNomeNodoCnf())) == false){
                          risultati.add(nuovo);
                          break;
                      }

                  }

              }
              else{
                  // this path includes the Node that the train needs to avoid,
                  // so it is not included in the list
              }

          }
      }

      return risultati;

  }
```