

Parallel execution of complex tasks using a distributed, robust and flexible agent-based platform

David Sánchez, Ángel Rodríguez, Antonio Moreno

Intelligent Technologies for Advanced Knowledge Acquisition (ITAKA) Research Group

Department of Computer Science and Mathematics

University Rovira i Virgili

Av. Paisos Catalans, 26. 43007 Tarragona

{david.sanchez; angel.rodriguez; antonio.moreno}@urv.cat

Abstract

The increasing complexity of AI related tasks often requires lots of computational resources. Parallel computing and, more concretely, grid based approaches have been an enormous success in the last years, providing a cheap and scalable approach for tackling calculation intensive problems. This paper presents a novel, general purpose approach for parallel computing based on agents. Multi-agent systems provide features such as high level abstraction, flexibility and distributed nature that are especially suitable for dealing with complex and highly dynamic problems. A particular application of the developed platform for implementing a complex knowledge acquisition method is also introduced, analysing the performance improvement from a temporal point of view.

1. Introduction

Artificial intelligence applications involve quite usually the processing of large amounts of data or huge information sources, and the execution of complex analytical processes. In those cases, the computational capacity of a unique computer may not be enough, requiring the use of multiple-CPU supercomputers or a computer network. This last approach, typically called *grid computing*, has been quite a success in the last years. This computing paradigm allows taking profit from unused computers, obsolete equipment or underused intranet nodes. This results in a reduction of the cost that typically implies parallel execution, configuring a highly scalable approach. In addition, Internet connections allow users to contribute with their own computers to world-wide scale distributed projects based on grid computing [2].

Grid-based applications are typically very specific low level programs performing repetitive tasks over large amounts of pre-processed data. In general, they are quite complex to develop, requiring the design of ad-hoc non-reusable execution frameworks.

In the last few years, agents and multi-agent systems [9] have appeared as a new promising computer engineering paradigm. On the one hand, agents provide a high level approach for implementing complex systems. On the other hand, multi-agent systems (MAS) provide an environment in which several entities can be transparently executed in a highly distributed and flexible manner. MAS also provide an added value thanks to features such as elaborated communicative skills and mobility capabilities.

This paper presents the design and implementation of a novel, high level, general purpose, flexible and robust platform for the parallel execution of tasks over a computer network using mobile agents. This last aspect is especially interesting because, as far as we know, very few attempts of agent mobility have been implemented. The platform provides an efficient framework in which execution tasks can be easily modelled over individual agents that are transparently managed and disposed over network nodes using an adequate load balancing policy.

The full system has been implemented with Java and using the JADE framework [3]. This configures an OS and hardware independent solution. The developed platform has been used as a testbed for a complex Web based knowledge acquisition system. It consists on a series of learning methods which crawl the Web analysing thousands of web resources in order to construct domain ontologies in a completely unsupervised and automatic way. Due to its complexity, it requires large computing resources and presents a dynamic and nondeterministic execution. Those

characteristics fit very well with the presented MAS platform, showing its potential usefulness in tackling real world problems.

The rest of the paper is organised as follows. Section 2 describes the design of the proposed agent-based parallel execution platform, including the physical topology, main components, management and configuration possibilities. Section 3 introduces the specific problem (domain ontology learning) to be adapted over a parallel environment, discussing its main characteristics and task behaviour. It also discusses the temporal cost and introduces the benefits that the proposed high level parallel implementation can bring over the sequential execution. The final section presents the conclusions and lines of future work.

2. Distributed agent platform

The proposed MAS environment has been designed to be very flexible and generic, providing an execution environment in which tasks can be distributed and executed in parallel in a transparent way. The only limitation is that concurrent tasks should be independent as communication between concurrent tasks is not supported at this moment.

Topologically, as shown in Figure 1, the platform is organised in a server side that should reside in a particular computer with known IP address and a set of client nodes in which tasks will be executed in parallel (belonging to one or several intranets or even disposed through Internet). The final user may access the system to request task executions and obtain final results via a Web based interface provided by the server.

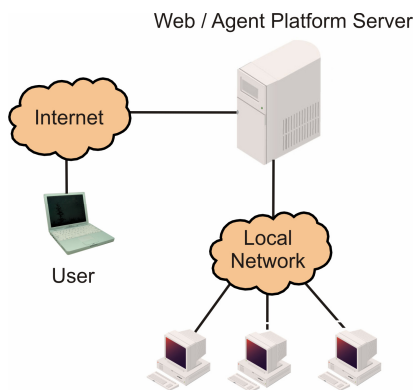


Figure 1. Basic agent based platform topology.

The server mission is to monitor the available client nodes and to initiate, distribute and finalize the agents that will execute tasks. Client nodes can be incorporated dynamically at any moment by registering themselves into the server, providing local information about their hardware characteristics (e.g. available RAM, CPU characteristics, architecture, etc.). They will host the agents that will execute the tasks requested by the server at each moment.

2.1. Platform components

The platform is composed by several agents that have been designed to provide a series of services required to manage a distributed system:

- *Grid Manager Agent* (GMA): it is located in the server and offers a registering service for client nodes and manages the execution of tasks by creating mobile agents.
- *Event Manager Agent* (EMA): it is also located on the server and constantly monitors agent events across the full platform. It can be aware of agent or nodes failures, allowing to implement error recuperation measures.
- *Registry Agent* (RA): it is executed in the client side, in order to register a particular node into the server, by specifying the client's hardware configuration.
- *Working Agent* (WA): it is a generic component that hosts a particular task. It is dynamically created by the GMA and associated to the task to be executed. Then, it is sent to a free client node assigned in function of the defined load balancing policy. It implements mobility capabilities in order to travel from the server side to the client nodes to execute tasks and return the results.
- *Node Manager*: it manages the computational resources available in the system at each moment. It performs the scheduling by means of a load balancing policy in order to select, at each moment, the most adequate client node in function of the task's requirements.
- *ID Manager*: it works as a name service, providing unique identifiers to WA.
- *Request Server*: it provides a service for receiving new tasks to be executed. By means of this component and an appropriate interface (web-based), the user can specify at each moment, which tasks should be executed.

2.2. Platform management

When the server module is deployed and executed in a computer, the JADE environment and the platform components described in the previous section are initialised. From that moment, the server is able to receive tasks execution petitions from the user. Those tasks are stored in a queue and assigned to the available client nodes.

Meanwhile, a set of client nodes should be setup. This can be performed dynamically at every moment of the platform lifecycle, providing a great degree of flexibility. Concretely, each new client node executes a setup process in which its own hardware configuration is inspected and this information, along with a registering petition, is sent to the server. An empty agent container is created at each node in order to allow future hosting of WAs. Once the server is aware of the node availability, it will start to send tasks to be executed on that computer.

Each task is defined as an object that encapsulates its characteristics, final results and even hardware requirements. They are assigned to client nodes using a scheduling policy that takes into consideration the hardware available at each free node, in order to provide an adequate load balancing. Once the server decides to execute a particular task on a specific node, a new WA is initialised and configured according to that task. The agent's unique name is assigned by the ID Manager. Then, the agent travels across the network, bringing the task request, task characteristics, execution state and specific source code with it. In this manner, all the components required to execute each process in the client node are received within the WA itself. This behaviour is very useful as it allows a very flexible mechanism for platform management as only *one* copy of the task(s) source code is needed in the server side. In this manner, task code can be updated easily and transparently to the rest of the platform. This supposes an added value to the platform not typically provided by typical grid-based executing environments.

WA mobility has been implemented using the advanced mobility capabilities provided in the last versions of the JADE framework. In the developed system, only intra-platform mobility is allowed but since JADE's 3.4 version, inter-platform mobility is also supported.

When the task has been executed at the client node, the result (typically a data file) is returned to the server that can present it to the user. It is important to note that a particular node can host several tasks (and the associated WAs) if enough hardware resources are available (according to the scheduling policy implemented by the Node Manager). In that case, we consider that a particular node provides a certain number of executing *slots*.

2.3. Event management

Any distributed system needs a mechanism to be aware of the incidences occurred in the platform.

In our case, as introduced in section 2.1, the EMA is able to monitor the state of platform component. It uses JADE's internal event mechanism in order to monitor, asynchronously, low level agent management messages sent by the JADE execution environment. In this manner, it is able to detect whenever a particular node, agent container or agent has failed. This is a very useful information, allowing the server to be aware, at each moment, about the state of client nodes. More concretely, node failure events, in conjunction with the dynamic registering service described in section 2.2, configures a very flexible mechanism for adding and removing client nodes to the platform.

Whenever a client node fails, crashes or shuts down, it may host, at that moment, one or several tasks. In this situation, it is necessary to implement a fail recovery mechanism to ensure the correct finalisation of tasks. In more detail, the server stores the status of each task-node-agent-container assignation. Whenever a node fails and its corresponding agent container does not respond, the system is aware of which task or tasks were executing in that location. Those tasks are put again in the execution queue and appropriately assigned to a new available node.

2.4. Configuration and interaction

Until now, we have presented an overview of the agent based platform. We have described how the task source code is stored on the server side and sent and executed according to user's requests.

However, in order to start executing tasks, the platform should be configured. Its generic design

allows adapting its behaviour in function of the tasks nature or the user preferences. Concretely:

- *Task definition*: each task should be specified by means of an abstract class *job*, that defines the main task's characteristics such as input and output parameters associated to the provided source code. The only restriction that tasks should fulfil in order to be correctly managed is to be mutually independent as inter communication between concurrent tasks is not supported.
- *Scheduling policy definition*: tasks are assigned to free node's executing slots in function of the available hardware. The specific criteria followed to define that assignation (the number of slots of each node and the prioritization of nodes) are specified by the Node Manager. A default policy considering typical hardware requirements is provided but, if desired the user may overwrite the corresponding class in order to adapt the system's behaviour to the tasks' hardware requirements. This allows a fine tuning of the scheduling process and an optimum use of the available hardware.

Task requests are meant to be specified via a web interface that provides a persistent environment in which user's profile, task results and task status are stored. This component can also be adapted to the particular problem's casuistry.

3. Case of study: Domain ontology construction from the Web

Once the generic design of the distributed execution platform has been presented, in this section we describe the implementation of a specific problem: a Web based domain ontology construction system. Its complexity and execution requirements make the application of the proposed MAS very suitable.

In the following sections, we will present the different aspects that have been taken into consideration to model and adapt it to the distributed platform (i.e. task analysis, computation complexity, task modelling, scheduling policy, etc.) and the benefits achieved in terms of performance from the runtime point of view.

3.1. Problem description

Ontologies [8] allow the definition of standard machine readable representations of knowledge. They are crucial components in many knowledge intensive areas like the Semantic Web [1], knowledge management, and electronic commerce. The construction of domain ontologies relies on knowledge engineers, which are typically overwhelmed by the size and complexity of a specific domain. In this sense, *automated ontology learning* methods allow a reduction in the time and effort needed to develop ontologies.

We have developed a method for domain ontology learning [6][7] using the Web as the source of knowledge. In a nut shell, it uses several knowledge acquisition techniques to extract domain concepts and relations from the analysis of thousands of web resources. Web corpus is retrieved by means of a standard web search engine that is also used to compute web-scale statistics about information distribution. Due to its unsupervised and automatic nature, as summarized in Figure 2, the learning process is divided in several steps that are iteratively executed as new knowledge is retrieved.

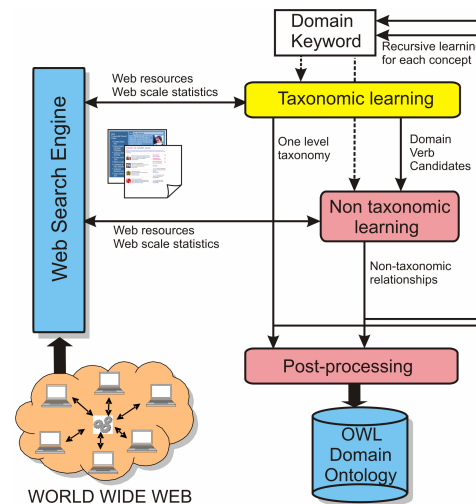


Figure 2. Overview of the domain ontology learning.

The process begins with a keyword that defines the domain to explore (e.g. *cancer*). Its analysis results in new taxonomically (e.g. *breast cancer*) and non-taxonomically (e.g. *cancer receives radiotherapy*) related concepts. Those are

recursively used as seeds for further analyses, composing a multilevel semantic structure. The system implements a self-controlling mechanism using feedback information about the learning to decide which branches to explore and when to stop the analysis. This results in a multilevel tree-like expansion as shown in Figure 3.

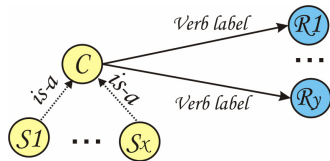


Figure 3. Learning expansion of the concept C with x taxonomic relations and y non-taxonomic relations.

At the end, the system is able to retrieve and present, in an ontological way, a set of domain terms, taxonomic and non-taxonomic relationships and even named entities.

Considering the enormous size of the Web and the unsupervised nature of the method, the degree of computational effort required to iteratively query web search engines and access and analyse thousand of web sites can be quite considerable. Certainly, not only CPU power but also internet bandwidth and RAM (needed to store pattern recognition files) are required. In consequence, as shown in Table 1, using a sequential implementation with one computer, the runtime needed to learn a domain ontology may take several hours.

Table 1. Summary of results obtained for several domains: number of taxonomic and non-taxonomic concepts discovered, web accesses and total runtime on one computer.

Domain ontology	#Taxonomic	#Non-taxonomic	#Web queries	Runtime
insect	668	236	58286	10 hours
cpu	134	121	13934	6 hours
tea	236	1430	57148	17 hours

After an empirical study, we have observed that the main influencing factor in the performance is the number of web accesses (mainly web search engine queries). This time is, in general, several orders of magnitude higher than the time needed for text analysis. In addition, web servers and search engines introduce overheads when consecutive accesses from the same machine are performed. In fact, as shown in

Figure 4, the runtime depends linearly on the number of queries performed to search engines.

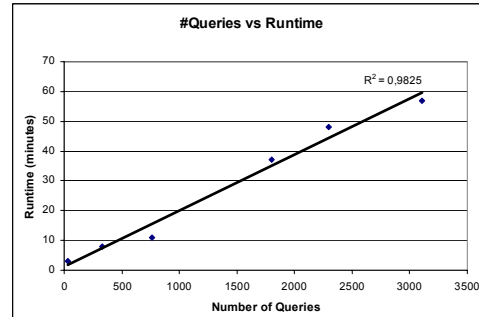


Figure 4. Runtime depends linearly on the number of Web search queries.

Considering the tree-like learning expansion behaviour presented in Figure 3, being T the runtime for a particular concept, the final runtime on one computer where ontological terms are sequentially evaluated is a *polynomial* function:

$$Time = T(\text{taxo_terms} + \text{notaxo_terms})^{\max(\text{taxo_depth}, \text{notaxo_depth})}$$

It depends on the number of taxonomically and non-taxonomically related concepts retrieved at each iteration. The exponent is the maximum depth of the relationships (typically the taxonomic depth will be higher but inferior to 4). The runtime (T) required to perform the analysis of a concept depends linearly on the web queries for statistics that, at the same time, depend linearly on the number of retrieved concepts. Considering the orders of magnitude managed (time in *minutes* and number of concepts in *hundreds*) one can easily realize that a sequential execution in one computer is not computationally feasible.

3.2. Task modelling

Taking into consideration the presented tree like expansion, several tasks (different analyses for each new concept) can be performed concurrently and independently. This workflow is adequate for the MAS-based parallel computing model.

In our case, the parallelisation benefits are not only related to the computational power, but also to other resources such as the Internet bandwidth or system memory. However, the most important aspect is that the parallel execution of various learning instances through several computers can

reduce the overhead of Web access, minimizing the execution waits and web search engine restrictions thanks to the distributed access from, ideally, different IP addresses.

In order to adapt the ontology learning execution to the proposed agent based platform, and thanks to its generic and high level design, we have only defined the following components:

- Each learning step has been modelled as a task, by extending the mentioned abstract class *job* with the appropriate input (parameters about search conditions, already acquired knowledge) and the output (files representing partial ontologies). The class constructor is associated with the source code of each ontology learning step.
- The scheduling policy has been overwritten according to the tasks' requirements. Considering that our computers have the same internet bandwidth, the differential factors are the CPU and the amount of available RAM. This last one is the most limiting factor as each task requires at least 100 MB due to the size of patterns files loaded to perform linguistic analyses. In consequence, we have designed a scheduler that defines and prioritizes free slots on available client nodes according to the amount of available RAM.
- In order to allow user interaction, a Web based interface has been designed. It allows specifying domains to explore, request the execution of learning tasks (associated to discovered concepts) and the visualization of partial results. The user also has information about the number of free slots and the tasks currently running. The web interface implements persistence mechanisms in order to store the state of partially executed tasks and previous results.

3.3. Parallel execution performance

Once the parallel modelling of the domain ontology learning approach has been presented, in this section we offer an analysis of the performance obtained using different degrees of parallelism. In this manner, we intend to show the benefits and the potential improvement from the runtime point of view that the parallel MAS platform provide over the non parallel approach for the specific problem (briefly introduced in section 3.1).

The first test consists in picking up four tasks of similar complexity (4 immediate subclasses of the *Cancer* domain) and to execute them, using the same parameters, in the following hardware configurations (see Table 2):

- 1 node with a unique free slot: each task will be modelled by a WA and executed sequentially. The final runtime is computed by adding each individual runtime.
- 2 computers with a unique free slot on each one: 2 tasks are modelled over a WA and sequentially executed in one computer in parallel with the other pair of tasks. The time is the maximum of both sequential executions.
- 4 computers with one slot on each one: each task is modelled over a WA that is executed in parallel. The final time is the maximum of the four executions.

Table 2. Performance tests for the execution of 4 similar learning tasks with different parallel conditions. Individual and total runtimes are presented.

Domain	1 node	2 nodes	4 nodes
Breast cancer	1083 s.	1093 s.	1095 s.
Lung cancer	980 s.	992 s.	1029 s.
Colon cancer	627 s.	667 s.	705 s.
Ovarian cancer	715 s.	812 s.	841 s.
Total	3405 s.	2085 s.	1095 s.

One can see that the improvement is, as expected, very significant and proportional to the degree of parallelism (see Figure 5). It is also interesting to note that the execution overhead introduced by the agent and platform management is negligible in relation to the sequential approach. This is due to the complexity and heavyweight nature of the executed tasks.

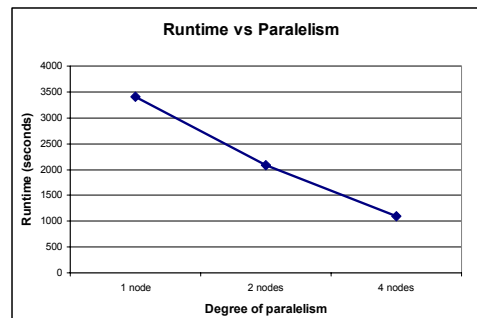


Figure 5. Performance increase in relation to the degree of parallelism.

The following test covers the full parallel execution of a complete example. In this case, we select one domain and executed 2 taxonomic levels sequentially (using one computer) and in parallel (using 4 computers) with automatic distribution of the work load in function of the implemented load balancing policy. From the performance obtained, we can check the degree of parallelism one can expect from the particular case and the scheduler's behaviour.

Using a wide domain (*Cancer*) and executing the taxonomic learning for the first two iterations, the results are 49 immediate subclasses that should be analysed. This process takes, in one computer, a total of 16505 seconds. Performing the same execution in parallel with 4 computers (a unique slot for each one), the total runtime is lowered till 5634 seconds. This supposes a performance improvement of 292% with a hardware increase of 400%. Examining the execution trace and representing the task-node assignation at each moment, we can compose the Gantt diagram shown in Figure 6.

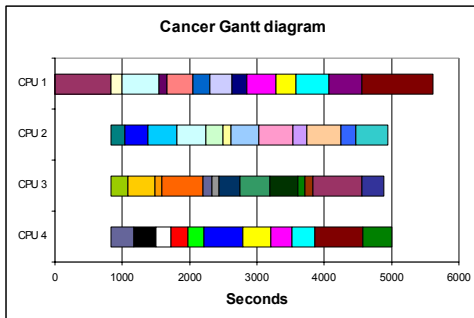


Figure 6. Distribution of taxonomic learning tasks among 4 computers for the *Cancer* domain.

One can see that the factor that compromises the parallel performance is the non-parallel interval at the beginning (necessary to obtain a first set of concepts to analyse) and at the end (needed to finish the latest task). In consequence, the potential improvement of this parallel approach is higher as more tasks (concepts) are available. In a complete learning process (involving hundreds of multi-level taxonomic and non-taxonomic analysis) the percentage of fully parallel execution interval will be much higher in relation to the sequential parts, and the throughput improvement percentage will tend to be similar to the hardware resources provided. As shown in the

first test of this section, the overhead introduced by agent and platform management is negligible in relation to the size of the tasks to execute.

Regarding the runtime required for each task, as stated in section 3.1, it depends linearly on the number of queries for statistics performed to the web search engine (see Figure 7). In this case, however, there is more variability due to the higher degree of parallelism.

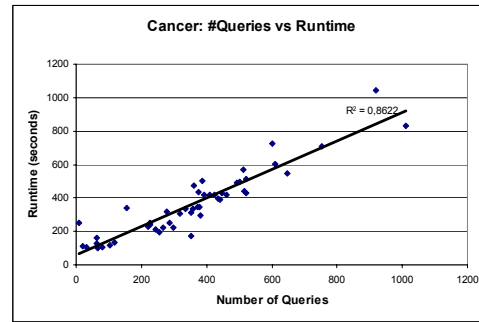


Figure 7. Number of queries vs. runtime for each learning task (subclass) of the *Cancer* domain. A linear dependence can be inferred.

Considering this execution environment, we can compare it to the sequential approach from a temporal point of view. As presented in section 3.1, the temporal cost in a sequential approach is a polynomial function of the number of concepts retrieved at each iteration multiplied an amount of times defined by the maximum depth of the discovered relationships. Without considering the limitations introduced by the available hardware, in the distributed approach we can parallelise the full set of retrieved concepts, reducing the runtime of one iteration to T (the time required to evaluate one concept). At the end, we are able to obtain a runtime of T^{max_depth} , where the exponent is the maximum depth of the taxonomic and non-taxonomic relationships (among 2 and 4). In consequence, we can reduce the time from $T(taxo_terms + notaxo_terms)^{max_depth}$ to T^{max_depth} using a $(taxo_terms + notaxo_terms)$ degree of parallelism.

In the real world, however, it is very unlikely to have such an amount of hardware and, in consequence, the real runtime will depend on the maximum degree of parallelism that we are able to achieve. Although one computer may host several tasks (in function of the executing slots defined by the node manager), in our tests we

have determined that one node with enough hardware resources (i.e. 2 Gigs of RAM, Pentium4 CPU or later) is able to execute among 6 to 8 tasks (and WAs) before the performance is degraded due to the concurrence overhead.

At the end, with a moderate amount of hardware, for this particular case, the parallel performance increase can suppose an improvement of one order of magnitude (from hours to minutes).

4. Conclusions

Agent technologies have been extensively used in the past for modelling complex systems [5]. Nowadays, they represent a mature technology that has been considered as the latest software engineering paradigm [4]. Certainly, they offer several characteristics that suppose an advantage.

One of the fundamental characteristics of multi-agent systems is their distributed nature. We have extensively used this feature to design and implement a novel environment for parallel computing. It has also been extensively tested using a complex real world problem for large scale Web knowledge acquisition. Performance results are quite impressive, considering the high level nature of our approach.

Additional benefits offered by the designed platform are:

- *Flexibility*: nodes can be added or removed from the platform at runtime. The system continuously adapts its behaviour to the available -potentially heterogeneous- hardware resources.
- *Scalability*: the performance scales linearly with respect to the number of nodes available. Those can be added easily with independence of its architecture or OS.
- *Robustness*: it implements fail safe measures, by constantly monitoring the platform state. Unsuccessful tasks are automatically reassigned to available nodes in a transparent way, ensuring a correct finalisation.
- *High level nature*: the use of agents and object oriented programming provides a high level environment that can be easily configured.
- *Genericity*: components have been designed in a general purpose manner, allowing to model different tasks easily and to adapt the system's behaviour accordingly.

As future lines of research we plan to study the inter-platform mobility offered by the latest version of JADE. This will allow the definition of several linked platforms, implementing non-centralized approach that will offer server replication and a higher degree of robustness. We will also consider the possibility of allowing communication between parallel tasks in order to improve the flexibility, allowing concurrent execution of dependant tasks. Finally, other complex and computing intensive problems will be modelled and analysed in order to test the performance improvement.

Acknowledgements

This work has been supported by the "*Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya i del Fons Social Europeu*".

References

- [1] Berners-lee, T., Hendler, J. and Lassila, O.: The semantic web. Scientific American. 2001.
- [2] GRID Infoware. Grid Computing Info Centre. <http://gridcomputing.com>
- [3] JADE. Java Agent Development Framework. <http://jade.cselt.it>
- [4] Jennings, N.: On agent-based software engineering. Artificial Intelligence 117. 2000. 277-296.
- [5] Moreno, A., Valls, A., Isern D. and Sánchez, D.: Applying Agent Technology to Healthcare: The GruSMA Experience. IEEE Intelligent Systems 21(6). 2006 63-67.
- [6] Sánchez, D. and Moreno, A.: A methodology for knowledge acquisition from the web. Journal of Knowledge-Based and Intelligent Engineering Systems 10(6). 2006. 453-475.
- [7] Sánchez, D. and Moreno, A.: Discovering Non-taxonomic Relations from the Web. In Proceedings IDEAL 2006. LNCS 4224. Burgos, Spain. 2006. 629-636
- [8] Studer, R., Benjamins, V.R. and Fensel., D.: Knowledge Engineering: Principles and Methods. IEEE Transactions on Knowledge and Data Engineering 25(1-2). 1998. 161-197.
- [9] Wooldridge, M.: An Introduction to multiagent systems. West Sussex, England: John Wiley and Sons, Ltd. 2002.