Agents in Grid system—design and implementation

Katarzyna Wasielewska[1], Micha Drozdowicz[1], Pawe Szmeja[2], Maria Ganzha[1], Marcin Paprzycki[1], Ivan Lirkov[3], Dana Petcu[4], Costin Badica[5]

[1] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
(maria.ganzha,marcin.paprzycki)@ibspan.waw.pl
[2] Technical University of Warsaw, Warsaw, Poland
[3] Bulgarian Academy of Sciences, Sofia, Bulgaria
[4] West University of Timisoara, Timisoara, Romania
[5] University of Craiova, Craiova, Romania

Abstract. We are developing an agent-based intelligent middleware for the Grid. It is based on agent teams as resource brokers and managers. Our earlier work resulted in a prototype implementation. However, our recent research led to a complete redesign of the system. Here, we discuss the new and main technical issues found during its implementation.

1 Introduction

The *Agents in Grid* (*AiG*) project aims at utilizing teams of software agents as resource brokers in the Grid. The initial overview of the approach can be found in [9], while the two main scenarios: (1) agents seeking teams to execute job(s), and (2) agents attempting to join the team, were discussed in [14, 13]. Since one of the main assumptions was that the system will use ontologies and semantic data processing, we have developed an ontology of the Grid (presented in [11]). While the initial work led to a demonstrator system, our recent studies led to a conclusion that redesign of the system is required, to assure more flexibility and use technologies that are rapidly maturing. The aim of this note is to present the new design, and outline main technical issues found during its implementation. To this effect, we start with an overview of the proposed system. In what follows, we discuss issues involved in the implementation of the redesigned system, moving from the front-end to the negotiation module.

2 System overview

In our work, Grid is considered as an open environment in which *agents* representing *users* interact to either (a) join teams, or (b) find teams to execute job(s). The infrastructure of our interest includes heterogeneous, geographically distributed, multi-domain computer resources. For such infrastructure it was always assumed (see, [12]) that it will become a source of income to its owners (e.g. in a manner similar to the Sun Grid by Oracle [8]).

In [9] we have outlined the system based on the following tenets (for more details, see the Use Case diagram and the discussion presented in [17]):

– agents work in teams (groups of agents)
– each team has a single leader—*LMaster agent*
– each *LMaster* has a mirror *LMirror agent* that can take over its job
– incoming workers (*worker agents*) join teams based on user-criteria
– teams (represented by *LMaster*s) accept workers based on team-criteria
– each *worker agent* can (if needed) play role of the *LMirror* or the *LMaster*
– matchmaking is facilitated by the *CIC* component.

Let us now briefly focus our attention on interactions between components of the system, i.e. the *User* and its representative the *LAgent*, and agent teams represented by *LMaster* agents (more information can be found in [9]). Let us assume that team "advertisements" describing: (1) what resources they offer, and (2) characteristics of workers they would like to "hire," are posted with the *Client Information Center* (*CIC*). Here, we concentrate our attention on two main scenarios in the system: *User* is looking for a team (1) to commission job execution, or (2) to join (to be paid for use of her resources). In both cases, the *User* interacts with her *LAgent*, and formulates conditions for (1) job execution, or (2) team joining. The *LAgent* communicates with the *CIC* to obtain a list of teams that satisfy these criteria. Next, the *LAgent* communicates with *LMasters* of selected teams, and utilizes the *FIPA Iterated Contract-Net Protocol* [2, 17] to negotiate the contract. If the *LAgent* finds an appropriate team, a *Service Level Agreement* (*SLA*) is formed. If no such team is found, the *LAgent* informs the *User* and awaits further instructions. Let us now present the new design of the system, outlining of the initial implementation, and proposed solutions to specific technical problems.

3 Ontologies in the system

As stated above, we assume that all data in the system will be ontologically demarcated. Therefore, we needed a robust ontology, covering concepts ranging from descriptions of hardware and software, through grid structure, to the *SLA* and contract definitions. After a comprehensive investigation of existing grid ontologies (see, [10]) we decided use the *Core Grid Ontology* (*CGO*, [19, 1]). While the *CGO* provided us with excellent base-terms concerning grid resurces and structure, we had to extend it to include the remaining concepts. The complete description of the ontology can be found in [10, 11]. The extended *CGO* (the *AiG Ontology*) is structured into three layers (its core classes depicted in Figure 1):

1. *Grid Ontology*—directly extending the CGO concepts.
2. *Conditions Ontology*—includes classes required by the SLA negotiations (e.g. pricing, payment mechanisms, worker availability conditions, etc.); it imports the *Grid Ontology*, to use terms related to the Grid structure and resources.

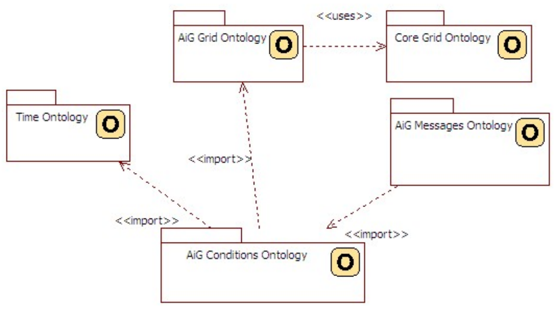


Fig. 1. Ontology diagram for AiG ontologies

3. *Messaging Ontology*—contains definitions of messages exchanged by the agents, forming the communication protocols of the system (uses the *Grid Ontology* and the *Conditions Ontology* to specify content of messages).

The crucial aspect of ontological modeling was representation of constraints.For example, when a user is looking for a team to have a job executed, she needs to specify the necessary hardware configuration. In this case, the common way of assigning values to class properties is not enough, as we also need to specify minimum, maximum, and range conditions. After considering several approaches, we have settled on class expressions. Here, requirements are defined as a new class that restricts the set of individuals to these satisfying conditions on class properties. We can thus ask a reasoner to infer a list of individuals of the generated class and receive the ones fulfilling the constraints.

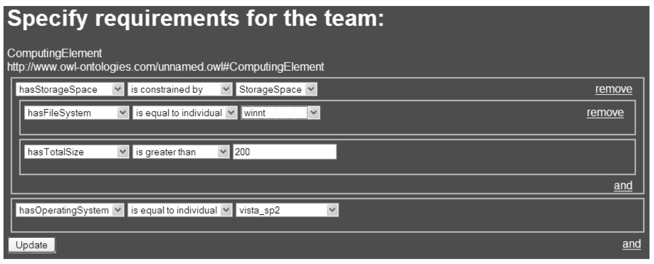
4 Negotiations in the system

Obviously, automated negotiations, and *SLA* management, are the key part of system. The *SLA*, is a result of negotiations, and defines agreement reached by the parties. Here, by negotiation we understand flow of messages between parties: *LAgent*s and *LMaster*s. As stated in [17], the negotiation process is based on the *FIPA Iterated Contract-Net Protocol*, and involves both *negotiable* and *static* parameters specified by the *User* through the front-end (described in the next Section). These parameters are passed to the *LAgent*, which forwards them to the *CIC*), which responds with list *LMasters* representing potential partners. Next, the *LAgent* construct a *Call-For-Proposal* message with an OWL instance of required resource description—for the job execution scenario; or of resource that *User* wants to sell—for the team joining scenario. The *CFP* contains also restrictions on contract conditions (for both cases). This message is sent to the selected *LMasters*, and those interested in the proposal reply with OWL instances representing their offers. Both parties shall use multicriterial analysis to evaluate received proposals and make offers that take into consideration their own ability to fulfill required conditions, as well as preferences.

5 Front-end design and implementation

In the current implementation, the front-end subsystem is mostly a means to provide ontological data for the negotiations. It consists of three parts, allowing specification of requirements concerning:

1. *Scheduling job execution*—lets *User* specify constraints on conditions of the contract regarding job execution. These include hardware and software requirements that a team has to satisfy in order to be taken into consideration.
2. *Joining a team*—specify information needed for negotiating joining a team, i.e. description of available resources, and restrictions on the contract.
3. *Worker acceptance criteria*—also concerns worker joining a team. Here, the owner of the *LMaster* can specify conditions that must be met by any worker willing to join the team. These may include hardware and software configuration, as well as terms of contract.


Specify requirements for the team:

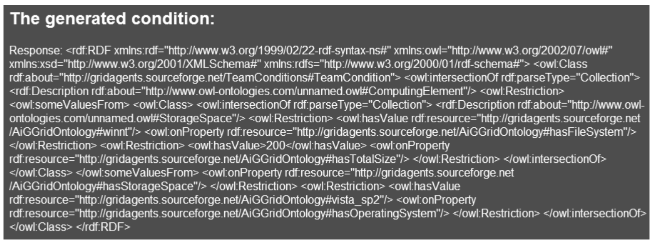
ComputingElement
http://www.owl-ontologies.com/unnamed.owl#ComputingElement

hasStorageSpace | is constrained by | StorageSpace
hasFileSystem | is equal to individual | winnt
hasTotalSize | is greater than | 200
hasOperatingSystem | is equal to individual | vista_sp2

Fig. 2. A condition builder section

In the initial system prototype. the front-end was a desktop application with the *LAgent* running in the background (on the same machine). Advantages of that approach included simple architecture, and ease of interactions between the client application and the agent. Hoever, this also meant that: 1) the *LAgent* could only work while the front-end application was running, and 2) at least part of the *User*'s data was stored on the local machine. Therefore, interacting with the *LAgent* from different machines would be difficult. Since the possibility of accessing an application from any computer becomes a necessity, we decided to develop a web application that can be hosted in a shared environment.

The core of the front-end is a condition builder—a set of condition boxes, each representing a description or constraint on a single class-property relationship (see, Figure 2). Depending on the selected class, the *User* may choose one of properties that the class is in domain of. Next, she can specify an *operator*, from within the applicable ones, to the selected property. For example, for datatype properties these may include: *equal to*, or *greater than*, whereas for object properties these would be *is equal to individual* and *is constrained by.*



The generated condition:

```
Response: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#" xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"> <owl:Class
rdf:about="http://gridagents.sourceforge.net/TeamConditions#TeamCondition"> <owl:intersectionOf rdf:parseType="Collection">
<rdf:Description rdf:about="http://www.owl-ontologies.com/unnamed.owl#ComputingElement"/> <owl:Restriction>
<owl:someValuesFrom> <owl:Class> <owl:intersectionOf rdf:parseType="Collection"> <rdf:Description rdf:about="http://www.owl-
ontologies.com/unnamed.owl#StorageSpace"/> <owl:Restriction> <owl:hasValue rdf:resource="http://gridagents.sourceforge.net
/AiGGridOntology#winnt"/> <owl:onProperty rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasFileSystem"/>
</owl:Restriction> <owl:Restriction> <owl:hasValue>200</owl:hasValue> <owl:onProperty
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasTotalSize"/> </owl:Restriction> </owl:intersectionOf>
</owl:Class> </owl:someValuesFrom> <owl:onProperty rdf:resource="http://gridagents.sourceforge.net
/AiGGridOntology#hasStorageSpace"/> </owl:Restriction> <owl:Restriction> <owl:hasValue
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#vista_sp2"/> <owl:onProperty
rdf:resource="http://gridagents.sourceforge.net/AiGGridOntology#hasOperatingSystem"/> </owl:Restriction> </owl:intersectionOf>
</owl:Class> </rdf:RDF>
```

Fig. 3. A condition builder section : OWL class

When an operator is selected, the system generates a fragment of user interface used to specify value of the property (see, Figure 3). Again, controls depend on the selected operator—be it a simple text box, or a drop down list of applicable individuals.

Interesting is the case of nested constraints. For object properties, when the *User* selects the operator *is constrained by*, for a class to be further specified, a new condition box is created within the existing one. It is used to describe the details, or requirements, regarding the value of the selected property. Front-end supports also setting constraints on multiple properties of the same class, using *and* buttons, which add a new condition box at the same level as the previous.

When the *User* finishes specifying conditions and pushes the *submit* button, the system parses the internal representation of conditions, and transforms it into an *OWL Class Expression* (see, Figure 3). This OWL fragment is passed to *GatewayAgent*, responsible for passing information between the application server, and the agent container. The *GatewayAgent* forwards the data to the *LAgent*, to handle it within the system.

In the front-end, all elements from which the *User* builds the ontological conditions and descriptions are generated dynamically, from the structure of the ontology. Therefore, all changes to the ontology can be applied automatically during system runtime. This is extremely important, especially in the case of ontology matching and enriching, based on information received from other agents. It also simplifies maintenance of changes in the ontology.

Furthermore, user interface elements are build dynamically, as responses to user actions. For example, if the *User* wishes to specify a particular CPU architecture, individuals of the *CPUArchitecture* class will only be fetched from the ontology when the *User* selects an *equal to individual* condition on the *hasArchitecture* property. This allows us to process only the needed parts of the ontology. Moreover, it allows to base displayed options on *User*'s previous choices. The could be a mechanism for providing automated assistance, by suggesting the most useful or common options, or by filtering out inconsisten options.

6 Passing ontological information

Communication in the system relies on extracting information from, and manipulating instances of, ontologies. Unfortunately, the default codecs for ontological data, found in the JADE framework, are very limiting in terms of what kind of ontological data can be transferred as a part of the message.

In our case we found it essential to be able to transfer arbitrary fragments of OWL ontologies, including TBox definitions of classes, used for representing constraints and requirements. This problem has been discussed in [16], and resulted in creation of the JADEOWL Codec [15]. Unfortunately, this plugin was extremely tightly integrated with the commercial RacerPro [7] reasoner, and its development seems to have stopped before the release of the OWL 2.0 specification. Therefore, we have developed our own JADE plugin, aiming at providing OWL support to the agent message processing.

Direct mapping of OWL 2.0 [4] into any static object-oriented programming language is not possible; i.e. there is no way to represent OWL as Java classes while preserving its dynamic structure and properties (partial solution to this problem can be found in [18]).Therefore, as opposed to the JADEOWL Codec, we have decided that any information instance, such as information about teams or negotiation deals, will be stored and accessed as OWL formatted text files. Thus, the plugin had to provide interface to files viewed both as raw text, and as OWL ontology; i.e. after passing a raw file, we had to be able to probe the structure of ontology, extract classes and instances, as well as their properties. In this way the plugin can serve as a high level interface to the structure and content of ontological messages, passed between JADE agents.

In any communication scenario, data is prepared in the form of OWL class or instance. For example, advertising a team by the *LMaster* involves sending an instance of an OWL class (describing the team) to the *CIC*, which recognizes it as a team advertisement and stores it in an OWL file. When asked by the *LAgent*, it filters all stored instances, to satisfy specified constraints.

Although messages contain raw OWL data, their interpretation is done internally by the plugin. In this way agents can access the information without the need to parse the text. This interpretation requires reasoning about the data, so an instance of a semantic reasoner is bundled with the communication plugin. Currently, the HermiT[3] reasoner is used. However, OWL API supports also other popular reasoners (e.g. Pellet[6], Fact++[5]).

To extract data (e.g. an OWL instance) from an ontology, a custom OWL class is created. The exact structure of the class depends on the data that needs to be extracted. For example, if the *CIC* is asked for agent teams with a Linux machine, it sends information received from the *LAgent* to the plugin. The plugin creates an OWL class that extends the definition of OWL class describing team advertisements, but also contains an OWL property restrictions that forces any instance of this class to be a team with a Linux computer. Other types of restrictions (like the cardinality restriction) supported by OWL 2.0[4] are also available.

Here, the reasoner performs consistency and satisfiability tests on the new class in the context of the ontology. If the tests fail, it means that the class cannot have any instances. An exception is thrown and the reasoner output is routed back to the creator of the instance, to inform about the problem and, possibly, how to fix it. After passing the tests, the class prepared in this way is presented to the reasoner that finds all its instances. The prepared OWL instances are sent back (as text) to the *LAgent* that requested the information.

Summarizing, the plugin aids creation of OWL classes and instances by producing and structuring the actual OWL text, while the reasoner (that is internal to the plugin) performs validity/consistency checks and filtering. This solution makes full ontological communication available, while preserving constraints set upon ontologies in OWL format.

6.1 Reasoning in team selection

After user specifies input data (e.g. worker's description, team members acceptance conditions, etc.), it is passed to the appropriate *LAgent*. The *LAgent*, interprets the received message and locally stores the information. To obtain list of candidate teams, message with the *OWL Class* represented as the raw OWL data describing criteria is constructed, and passed to the *CIC* (see, Section 6). Reasoning in the back-end part of the system is required for both negotiating parties, i.e. the *LAgent* and the *LMaster*s. In the proof of concept application, the *LAgent* utilized a linear-additive model for three predefined criteria, in an offer selection process. This model is a simple MCA model, therefore, in the redesigned system implementation we plan to use also other MCA methods to evaluate offers received by the *LAgent*. We are also going to consider additional criteria available in the ontology. On the other hand, *LMaster*s use MCA to determine e.g. cost of job execution. Each resource needed for job execution e.g. memory, bandwidth has a pricing property in the ontology that specifies pricing type and price. To evaluate total price of job execution, the *LMaster* combines prices for each required component.

Reasoning is also used by the *LMaster*s to verify if they are able to execute a given job i.e. if there is an available member in the team that has resources required to execute a job. So far, team members resource descriptions have been stored in *CIC* component, however, they will be stored also locally so that the *LMaster* can use reasoner on it's local ontological database.

7 Concluding remarks

The aim of this paper was to discuss an outline of the implementations and solutions applied to selected technical problems within the scope of the *AiG* project. Currently, we are proceeding with testing of the above described components of the system, i.e. the front-end web application, the back-end agent-based application, and a communication bridge between these components—the OWL plugin for ontology-based interactions.

Acknowledgments

Work of Maria Ganzha and Michal Drozdowicz was supported from the "Funds for Science" of the Polish Ministry for Science and Higher Education for years 2008-2011, as a research project (contract number N516 382434). The Polish-Bulgarian collaboration was partially supported by the *Parallel and Distributed Computing Practices* grant. The Polish-Romanian collaboration was partially supported by the *Agent-Based Service Negotiation in Computational Grids*; and the *Agents, Grids and Heterogeneous Computing* grants.

References

1. Core grid ontology. `http://grid.ucy.ac.cy/grisen/cgo.owl`.
2. Fipa iterated contract net interaction protocol specification. `www.fipa.org/specs/fipa00030/PC00030D.pdf`.
3. Hermit owl reasoner. `http://hermit-reasoner.com/`.
4. Owl 2 web ontology language. `http://www.w3.org/TR/owl2-overview/`.
5. Owl: Fact++. `http://owl.man.ac.uk/factplusplus/`.
6. Pellet: Owl 2 reasoner for java. `http://clarkparsia.com/pellet/`.
7. Racerpro 2.0. `http://www.racer-systems.com/`.
8. Sun utility computing. `http://www.sun.com/service/sungrid/`.
9. M. Dominiak, W. Kuranowski, M. Gawinecki, M. Ganzha, and M. Paprzycki. Utilizing agent teams in grid resource management—preliminary considerations. In *Proc. of the IEEE J. V. Atanasoff Conference*, pages 46–51, Los Alamitos, CA, 2006. IEEE CS Press.
10. M. Drozdowicz, M. Ganzha, M. Paprzycki, R. Olejnik, I. Lirkov, P. Telegin, and M.Senobari. Parallel, distributed and grid computing for engineering. chapter Ontologies, Agents and the Grid: An Overview, pages 117–140. Saxe-Coburg Publications, Stirlingshire, UK, 2009.
11. M. Drozdowicz, K. Wasielewska, M. Ganzha, M. Paprzycki, N. Attaui, I. Lirkov, R. Olejnik, D. Petcu, and C. Badica. Trends in parallel, distributed, grid and cloud computing for engineering. chapter Ontology for Contract Negotiations in Agent-based Grid Resource Management System. Saxe-Coburg Publications, Stirlingshire, UK, 2011.
12. I. Foster and C. Kesselman, editors. *The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure*. The Elsevier Series in Grid Computing. Elsevier, 2004.
13. W. Kuranowski, M. Ganzha, M. Gawinecki, M. Paprzycki, I. Lirkov, and S. Margenov. Forming and managing agent teams acting as resource brokers in the grid—preliminary considerations. *International Journal of Computational Intelligence Research*, 4(1):9–16, 2008.
14. W. Kuranowski, M. Ganzha, M. Paprzycki, and I. Lirkov. Supervising agent team an agent-based grid resource brokering systeminitial solution. In F. Xhafa and L. Barolli, editors, *Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems*, pages 321–326, Los Alamitos, CA, 2008. IEEE CS Press.
15. B. Schiemann. Jadeowl codec. `http://www8.informatik.uni-erlangen.de/en/demosdownloads.html`.
16. B. Schiemann and U. Schreiber. OWL DL as a FIPA ACL content language, 2006.

17. K. Wasielewska, M. Drozdowicz, M. Ganzha, M. Paprzycki, N. Attaui, D. Petcu, C. Badica, R. Olejnik, and I. Lirkov. Trends in parallel, distributed, grid and cloud computing for engineering. chapter Negotiations in an Agent-based Grid Resource Brokering Systems. Saxe-Coburg Publications, Stirlingshire, UK, 2011.
18. Y. Xin-yu and L. Juan-zi. Research on mapping owl ontology to software code model. 2009.
19. W. Xing, M. D. Dikaiakos, R. Sakellariou, S. Orlando, and D. Laforenza. Design and development of a core grid ontology. In *Proc. of the CoreGRID Workshop: Integrated research in Grid Computing*, pages 21–31, 2005.