Utilizing Agent Teams in Grid Resource Brokering*

M. Dominiak† M. Ganzha‡§ M. Gawinecki‡
W. Kuranowski¶ M. Paprzycki‡‖ S. Margenov** I. Lirkov**

June 1, 2007

Abstract

Recently it was suggested that (mobile) software agents can provide an infrastructure for resource brokering and management in Grids. In this paper we introduce a novel approach based on agent teams. Since the *yellow pages*-based approach was selected to facilitate resource discovery, we summarize results of our experiments to find an efficient ways of implementing *yellow page* service in an agent-based system. We also discuss how agents can find a team that will execute their job.

1 Introduction

Grid computing has emerged as a promising approach to utilizing heterogeneous, geographically distributed, multi-domain computer resources. Virtualization of computing resources by Grid computing is expected to provide its users with highly available and adaptable computing utilities. It is also expected to have a broad impact in science, businesses and industries. Unfortunately, the uptake

*Work of M. Ganzha, M. Paprzycki, S. Margenov, and I. Lirkov is sponsored by the Poland-Bulgaria Academy of Science cooperation grant.

†Technical University of Warsaw
‡Systems Research Institute of the Polish Academy of Science
§Elblag University of Humanities and Economics
¶Software Development Department, Wirtualna Polska
‖Warsaw School of Social Psychology
**Institute for Parallel Processing, Bulgarian Academy of Science

of the Grid, while speeding-up recently, is still unsatisfactory. One possible reason for this situation is an overly complicated support for resource brokering and management provided by current Grid software infrastructure.

At the same time, it has been suggested that software agents combined with ontologies may provide the necessary infrastructure, by infusing Grid with intelligence [8, 16]. Accepting arguments presented there, we have searched for the existing solutions that match this vision. While the results of our search are summarized in the next section, we can say that in our view all found solutions are somewhat limited in scope and robustness. Therefore we propose a different approach that is based on agent teams that collaborate to fulfill user requirements. In the next section we briefly summarize the state of the art in agent-based Grid resource management. We follow with the description of our proposed system. Next, we present how we have designed a resource discovery service and discuss how the proposed approach can be implemented efficiently. We complete the paper by discussing procedures involved in agents selecting team that will execute their job, and outlining future research directions.

2 Agents in Grids today

The initial work on agents in Grids can be traced at least to [2], where J. Cao and colleagues addressed the question of resource discovery in Grids. They proposed a hierarchical agent-based structure and experimentally evaluated various optimizing strategies for information distribution. Obviously, while very interesting, this work addresses only a small sub-area of usage of agents in Grids. Furthermore, the proposed framework was to be anchored in the PACE infrastructure, which by now seems to be extinct.

More recently B. Di Martino and O. Rana have proposed MAGDA (Mobile AGent Distributed Application), a mobile agent toolkit designed to support (1) resource discovery, (2) performance monitoring and load balancing, and (3) task execution within the Grid [6]. Here, a dedicated mobile agents visit servers in the Grid and collect system information (gathered by local static agents) that is used to optimize distribution of application workload among agents or to move it from a heavier loaded node to a less loaded one (computational tasks are carried by mobile agents to nodes where they will be executed). However, the proposed system does not have an economic model associated with it. Furthermore, it was implemented using Aglets agent environment which, though recently becoming an open source product, seem to be slowly turning into a historical reference.

In 2005, S. S. Manvi and colleagues proposed somewhat different approach to agents in Grids [13]. They started from an economic model and utilized mobile agents which traverse the network to complete a user defined task. At each visited

node agents find what are local conditions for job execution and if acceptable, execute their job there (if they are not, they move on). In their work, among others, authors consider a number of pathway selection scenarios.

Also in 2005, D. Ouelhadj and colleagues considered negotiation (and re-negotiation) of a Service Level Agreement between agents representing resources and resource users [14]. Negotiations were to be based on the Contract Net Protocol, however their paper was focused on higher level functionalities of the system. Again, this work considers only a specific sub-area of utilization of agents in Grids.

While interesting, we can see some problems with the proposed approaches. (1) Most of them are limited in scope and functionality and do not involve economical foundations. (2) Some of them rely on agent mobility, while not considering its cost — since agents carry tasks (and possibly data), their size depends on the size of transported code and data and thus agent mobility should be used very judiciously. (3) Proposed infrastructures do not take into account full effect of Grids highly dynamic nature and use single service providers — this leaves users vulnerable to potential rapid fluctuations of workload of individual nodes, as well as nodes disappearing and reappearing practically without warning. (4) Finally, reliance on "barely known" service providers should involve trust (reputation) management.

3 Proposed approach

Let us start from two assumptions that shape our proposed solution. The most basic one is that we envision the Grid to be an "open environment," which ultimately can consist of any computer connected to the Internet [8]. Therefore, we are less interested in "local Grids" that span a single laboratory or organization and thus can be strictly controlled by their administrators.

Second, from a pragmatic and functional perspective, we view the computational Grid as an environment in which workers (in our case *agent workers*) that want to contribute their resources, and be remunerated for their usage, meet and interact with users (in our case *agent users*) that want to utilize offered services to complete their tasks.

Taking these two assumptions into account it is easy to see that a **single worker**, for example representing a typical "home-user," has somewhat limited value. While we recognize success of applications like *SETI@home* that is based on harnessing power of millions of "home-PC's," this application (and a number of similar ones) has very specific nature. There, the fact that a particular resource "disappears" during calculations is rather inconsequential, as any data item can be processed at any time and in any order. Furthermore, data item that was not

completed due to the "vanishing PC" can be completed in the future by another resource. This, however, is not the case in business-type applications, where calculations have to be completed in a specific order and, usually, within a well-defined time-frame. In other words, in most applications some form of a service-level agreement (*SLA*), that assures conditions of job completion has to be involved. Assuring such *SLA* in the case of a "home-PC," is almost impossible. Therefore, to address this problem, we introduce virtual organizations, called *agent teams* that are based on the following general assumptions (for more details see [4]):

- 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 in case when it "goes down"
- incoming workers (*worker agents*) join teams based on individual set of criteria
- teams (represented by their *LMaster*s) accept workers based on individual set of criteria
- decisions about joining and accepting involves multicriterial analysis (performed by so-called *MCDM modules*)
- each *worker agent* can (if needed) play the role of an *LMaster* (and thus of an *LMirror*)
- matchmaking is provided through yellow pages [17] and facilitated by the *CIC* agent [1]

Combining these assumptions we can develop system represented in Figure 1 as a Use Case diagram.

Let us start from the observation that for an agent team to be visible to potential users or team members, it must post its *team advertisement* in an easily reachable way. As described in [17], there are many ways in which information used in matchmaking can be made available in a distributed system and each one of them has advantages and disadvantages. In our work we have decided to utilize a *yellow page*-type approach and thus *LMaster* agents post their team advertisements within the *Client Information Center* (*CIC*). Such advertisements contain both information about offered resources (e.g. hardware capabilities, available software, price etc.) and "team metadata" (e.g. terms of joining, provisioning, specialization etc.). In this way *yellow pages* can be used: (1) by *user agents* looking for

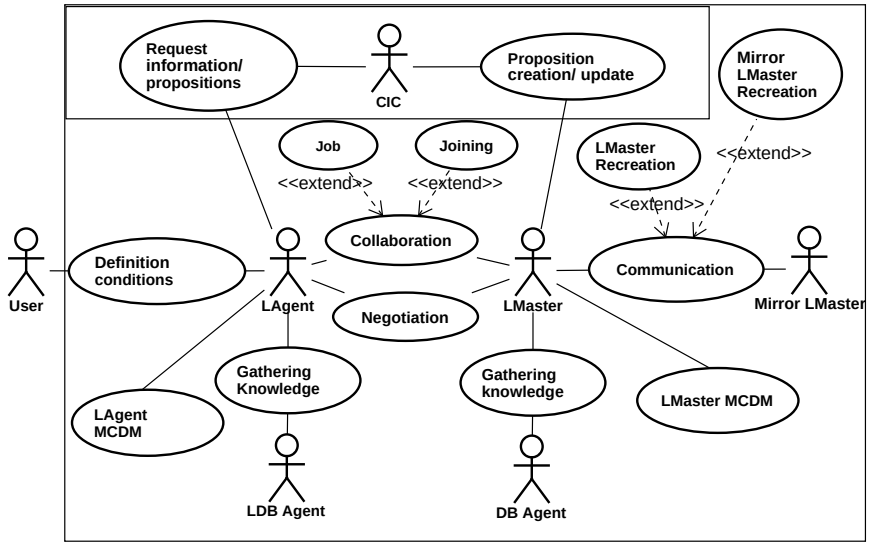

Figure 1: Use Case diagram of the proposed system

resources satisfying their task requirements, and (2) by *worker agents* searching for a team to join. For example, *worker agent* representing a computer with installed Matlab software, may want to join a team specializing in solving problems utilizing Matlab.

Let us now describe dynamic processes that are depicted in their static form in Figure 1. To do this let us assume that the system is already running for some time, so that there already exist agent teams and their "advertisements" (describing both resources they offer and agents they would like to join their team) are posted within the *CIC*. First, observe that the *User*, represented in Figure 1, can either be someone who tries to contribute services to the Grid or someone who would like to utilize services available there. Interestingly, the Use Case diagram shows that both situations can be modeled in a "UML-symmetric" way. Let us start from the case of *"User-contributor"* (processes that take place here are very similar to these described in [1] that provides further details).

User who wants to contribute resources to the Grid communicates with its agent (the local agent *LAgent* that becomes a worker agent) and formulates conditions for joining an agent team. Note that she may also request creation of a new team and her *LAgent* becoming its *LMaster*. The *LAgent* requests from the *CIC* list of agent teams that satisfy its predefined criteria. Upon receiving such a list, due to trust considerations (see [9] for more details) it may remove certain teams from the list. For instance, if it worked with a given team in the past and

was “unhappy” with “rewards,” it may not want to work with it again. For all the teams remaining on the list, the *LAgent* communicates with their *LMasters* utilizing FIPA Contract Net Protocol based negotiations [7] and multicriterial analysis [12] to evaluate obtained proposals. The result of interactions between the *LAgent* and *LMaster*s may be twofold: (1) it finds a team that it would like to work with and joins it, (2) no such team is found (either it is not interested in any offer from *LMasters* or no *LMaster* send an offer). In this situation the *LAgent* may decided to abandon the task and inform about it its *User*. It is also possible that the *LAgent* decides to become the *LMaster* of a new team itself. In this case, it prepares an offer describing (1) who it would like to invite to join its team, and (2) what resources it can provide to users (i.e. what kind of jobs it is willing to work on); and send these two “advertisements” to the *CIC* to be “posted.”

Let us now briefly consider what happens when the *User* requests that its *LAgent* arranges execution of a task (for in-depth discussion see section 5). The *User* specifies conditions of task execution (e.g. maximal price). The *LAgent* queries the *CIC* to find out which teams can execute its task. Upon receiving a list of teams that match the query, the *LAgent* removes from it teams that cannot be trusted. Next, it communicates with *LMaster*s of the remaining teams and uses FIPA Contract Net Protocol and multicriterial analysis to find the best team to execute its job. Note that if no team will satisfy conditions imposed by the *User* then no deal will be reached. In this case the *LAgent* will report this situation to its *User* and await further instructions.

Let us now describe the relationships between the *LMaster* and the *LMirror*. When a new team is created, then the “founding agent” becomes its *LMaster*. The first agent that joins that team becomes the *LMirror* (agent that can take over the team-lead in case when anything happens to the *LMaster*). Subsequent agents joining the team will become *worker agents*. We have not decided yet if the *LMirror* should be also working as a *worker agent* or if its role should be limited to mirroring the *LMaster*; this decision will based on experimental analysis of *LMirror*s workload and will be performed when the initial feature-complete version of the system will be implemented. The *LMaster* and the *LMirror* share all information that is pertinent to the existence of the team; e.g. list of workers and their characteristics, list of tasks that have been contracted and have to be executed, knowledge base that stores information about all past interactions with incoming users etc. It is assumed that the *LMaster* and the *LMirror* check each-others existence regularly in short time intervals. In the case when the *LMaster* does not respond to a ping-type ACL message the *LMirror* contacts the agent environment infrastructure (*Agent Management System* agent) to check the status of the *LMaster*. If the *LMaster* is “gone” it takes over its role. Its first action is to promote one of worker agents to become its *LMirror* and pass to it all necessary information. Then it informs all necessary agents about the change (the fact that it is now

the *LMaster* of the team). Similarly, the *LMaster* upon finding that the *LMirror* agent is "gone" immediately promotes one of worker agents to become *LMirror* and passes to it all necessary information. In both cases, promotion of a worker to a role of an *LMaster* or an *LMirror* may require dealing with the task that the selected worker was executing at the time of its "promotion." Let us note that the proposed solution is not bullet-proof. It is conceivable that both the *LMaster* and the *LMirror* will go down "almost simultaneously" (e.g. the *LMaster* realizes that the *LMirror* is gone, but before it promotes one of its workers to become its new *LMirror* it will go down itself) and thus the team will be "destroyed." However, such a situation should be relatively rare and our goal is not to create a completely bullet-proof infrastructure. Rather, our aim is to provide the proposed infrastructure with a reasonable level of resilience against common failures. Obviously, in a production environment further levels of defense against team destruction would have to be developed.

Finally, let us briefly mention a few additional objects that appear in Figure 1. The *Gathering knowledge* functions denote collection of information about processes happening in the system. The *LMaster* collects information about all interactions with incoming task-carrying agents as well as about members of its team. In this way it may later decide to not to interact with certain clients or remove certain workers from its team. Similarly, the *LAgent* collects knowledge about what happened when it utilized services of various teams, as well as when it was a worker for various teams. Interestingly, since *LAgent* can play any role in the system, it is quite possible that an *LMaster* will turn into an *LAgent* who represents its *User* trying to find location to execute its task. Will it turn to its own former team to do it? Questions like this are going to be answered within the *LAgent MCDM* module and the *LMaster MCDM* module.

4 Development of an efficient *CIC* infrastructure

4.1 *LAgent – CIC* interactions

As we have seen, regardless of the scenario, interactions with the *CIC* are crucial to the functioning of the system. Therefore, let us now discuss interactions that take place when the *LAgent* is querying the *CIC* where to execute its task.

We have assumed that data in our system is to be stored in semantically demarcated form. In this context, an ideal situation would be if there existed an all-agreed "ontology of the Grid." Unfortunately, while there exists a number of (separate and incompatible) attempts at designing such an ontology, at this stage they can be treated only as a "work in progress." Therefore, instead of selecting one of them and paying the price of dealing with a large and not necessarily fitting our needs ontology (which would then mean that we would have to make changes

in an ontology that we have not conceived and have no control over), we focus our work on the agent-related aspects of the system (designing and implementing agent system skeleton) while utilizing simplistic ontologies). Obviously, when the Grid ontology will be agreed on, our system *will be ready* for it. Currently, our ontology of Grid resources is focused on their "computational" aspects, e.g. processor, memory and available disk space. What follows is a snippet of this, OWL Lite based, ontology:

```
@prefix : <http://Gridagents.sourceforge.net/Grid#> .

:Computer
        :a owl:Class.

:hasCPU
        :a owl:ObjectProperty;
        rdfs:range :CPU;
        rdfs:domain :Computer.

:CPU
        :a owl:Class.

:hasCPUFrequency
        :a owl:DataProperty;
        rdfs:comment "in GHz";
        rdfs:range xsd:float;
        rdfs:domain :CPU.

:hasCPUType
        :a owl:ObjectProperty;
        rdfs:range :CPUType;
        rdfs:domain :CPU.

:CPUType
        :a owl:Class.

Intel :a :CPUType.
AMDAthlon :a :CPUType.

:hasMemory
        :a owl:DatatypeProperty;
        rdfs:comment "in MB";
        rdfs:range xsd:float;
        rdfs:domain :Computer.
```

```
:hasUserDiskQuota
        :a owl:DatatypeProperty;
        rdfs:comment "in MB";
        rdfs:range xsd:float;
        rdfs:domain :Computer.

:LMaster
  :a owl:Class;

:hasContactAID
        :a owl:ObjectProperty;
        rdfs:range xsd:string;
        rdfs:domain :LMaster.
```

Let us now assume that agent *LMaster007* has in its team worker *PC1410* which has a 3.5 GHz Intel processor, 1024 Mbytes of memory and 600 Mbytes of disk space available as a "Grid service." In our ontology it would be represented as:

```
:LMaster007
  :hasContactAID
        "monster@e-plant:1099/JADE";
  :hasWorker :PC1410.

:PC2929
        :a :Computer;
        :hasCPU
        [
                a :CPU;
                :hasCPUType :Intel;
                :hasCPUFrequency "3.5";
        ] ;
        :hasUserDiskQuota "600";
        :hasMemory "1024".
```

Ontologically demarcated data is stored (by the *CIC*) in a Jena 2.3 repository [11]. To query Jena persisted data we have decided to use the SPARQL language [15]. Let us now assume that the *LAgent* is looking for a computer (to execute its job) with an Intel processor of at least 3.0 GHz, at least 512 Mbytes of RAM, and at least 500 Mbytes of disk space. Then the SPARQL query will have the form:

```
PREFIX : <http://Gridagents.sourceforge.net/Grid#>
SELECT ?contact
WHERE
{
  ?lmaster
```

```
    :hasContactAID ?contact;
    :a :LMaster;
    :hasWorker
      [
        :a :Computer;
        :hasCPU
          [ a :CPU;
            :hasCPUType :Intel;
            :hasCPUFrequency ?freq;
          ];
        :hasUserDiskQuota ?quota;
        :hasMemory ?mem;
      ].
  FILTER (?freq >= 3.0)
  FILTER (?quota >= 500)
  FILTER (?mem >= 512)
}
```

and the response that points to the above described machine (which satisfies the search citeria) would look as follows: `monster@e-plant:1099/JADE`. Specifically, it points to the *LMaster* that has that machine (worker) in its team. Obviously, a complete response would consist of a list of all teams that have among them at least one machine that satisfies the above described criteria.

4.2 Efficient implementation of the *CIC*

As it should be obvious, the *CIC* infrastructure is one of the key components of our system. Since interactions between *user agents* and the *CIC* are the necessary part of early stages of preparing job execution, or *user agent* joining an agent team, long delays in responses from the *CIC* would become a bottleneck of the system. Therefore the *CIC* should efficiently handle large number of requests. Since our solution for *CIC* services was a centralized yellow-page approach, we have decided to find its optimal implementation. Here, we follow our earlier studies in efficiency of our agent platform of choice — JADE [10]. In [3] we have shown, among others, that the best performance in database queries was observed when a single agent received and enqueued client-requests, while multiple database accessing agents (*SQLAgents*) dequeued requests and executed them on the database. Furthermore, all (SQL) query-processing agents and the database run on separate computers and when five *SQLAgents* were used, performance gain of almost 33% was reported. We have followed this example and experimented with four different *CIC* architectures. The complete results can be found in [5]. Here we will report only the performance of two best architectures: (1) threaded,

and (2) architecture with distributed database querying agents and an additional *CIC Internal Agent* (*CICIA*).

In the threaded architecture we utilize the well-known task-per-thread paradigm. We use Java threads and make them accessible to the *CIC agent* within its container. Each worker thread has its own connection to the database and its instance of the Jena model. Initialization of these resources is computationally expensive and that is why instead of spawning new threads, we use preinitialized threads in the worker thread pool. The *CIC* agent picks requests (query-requests or yellow-pages-update-requests) from the JADE-provided *message queue*. Note that each JADE agent comes with its own message queue provided by the JADE environment. Furthermore, this queue is the only way for the *CIC* to receive ACL messages from other agents. The *CIC* agent extracts from the *message queue* all messages that require access to the database and enqueues them into another queue — the *request queue*, which we have implemented in Java. It is this *request queue* from which free worker threads pull requests for execution. After executing the query they send obtained responses to their originators. This architecture is depicted in Figure 2. In the second approach we use *CICDbAgents* —

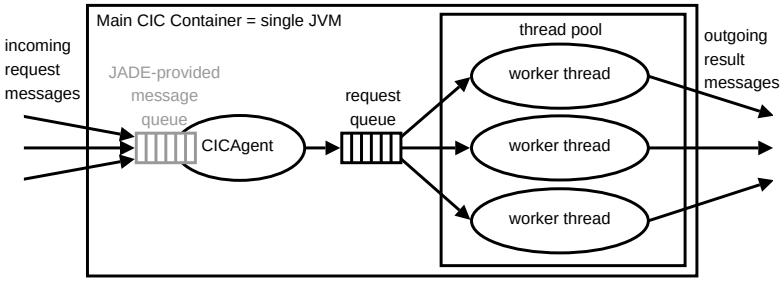

Figure 2: Request/result flow in a multi-threaded *CIC* architecture.

instead of worker threads. Each of these agents resides on a separate computer. Again, incoming messages are stored in the JADE-provided *message queue* of the *CIC* agent. As above, messages that request database access are removed from the *message queue* and enqueued into, implemented in Java, *request queue*. This queue acts as a buffer between the *CIC* agent and the *CICDbAgents* and, furthermore, reduces the number of messages stored in the JADE *message queue* (which size is limited only by local resources).

Incoming requests are **delegated** by the *CIC* agent (in the form of ACL messages) to "free" *CICDbAgents* (push-based approach). Each database agent completes one task (request) at a time and sends results to the *CIC Internal Agent* (*CICIA*). These results– send as ACL messages—are stored in the JADE *message queue* of the *CICIA*). The *CICIA* agent removes them from its *message queue* and enqueues them into a synchronized *result-queue*, from which they are dequeued

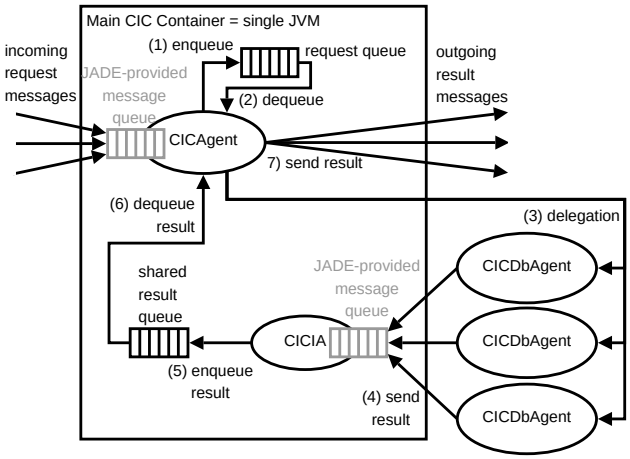

Figure 3: Request/result flow in distributed multi-agent *CIC* with *CICIA*.

by the *CIC* agent and send back to requesters. In other words, the intercommunication between the *CICIA* and the *CIC* agent is accomplished through a shared *result-queue* (instead of ACL messaging). As it is easy to see, this is also why (the *CICIA* and the *CIC* agent) must run within the same agent container (the *Main CIC Container)*. The proposed architecture is depicted in Figure 3. Let us observe that the *CIC* agent has three behaviors: (1) receive request-message from its *message queue* and enqueue it in the *request queue*, (2) dequeue request from the *request queue* and send it to a "free" *CICDbAgent*, and (3) dequeue message from the *result queue* and send it to the requester. The sequence diagram of handling the request is presented in Figure 4. In our experiments, to simulate a flow of incoming requests from *user agents* we used 4 *Querying Agents* (QA), requesting the *CIC* to perform SPARQL [15] resource queries. It should be noted that the form of the SPARQL query can change performance of the system. The ARQ engine used in Jena, is responsible for executing the query on OWL resources persisted in the database. It translates only parts of the SPARQL query into SQL. The remaining parts (e.g. FILTER operations) of the SPARQL query are not performed through the SQL query, but locally by the ARQ engine, utilizing local JVM resources. In our case queries had the following form:

```
PREFIX : <http://Gridagents.sourceforge.net/Grid#>
SELECT ?master
WHERE {
        ?comp :cpuClockSpeedMhz ?cpu.
        ?master :offersResource ?comp.
        FILTER (?cpu > 1000)
}
```

Each *QA* was running concurrently on a separate machine, and was sending $2,500$

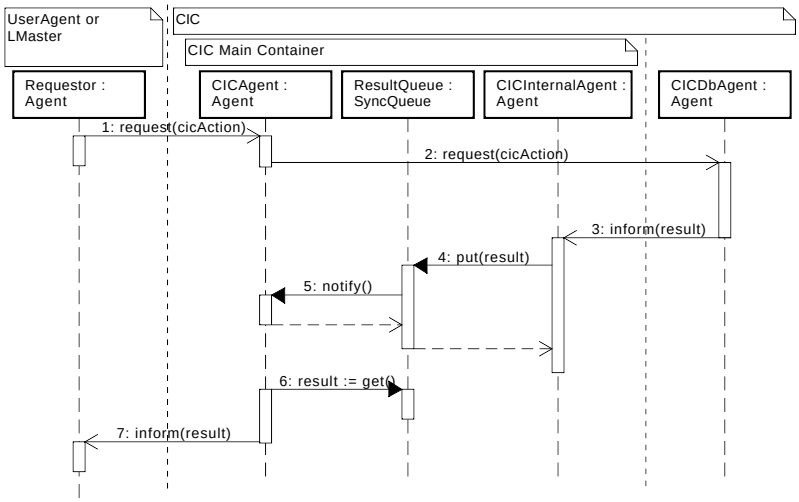

Figure 4: Sequence diagram: Handling requests by *CIC* with additional *CIC Internal Agent*.

requests and receiving query-results. Thus in each experiment $10,000$ queries have been processed by the *CIC*. Since we have been running multiple experiments (especially when attempting at performance tuning), we have developed an experimental framework for running tests automatically, while varying their parameters (e.g. number of worker threads, number of *CICDbAgents* etc.). All experimental runs were coordinated by the *Test Coordinator Agent* (*TCA*). Before each test, remote JADE agent containers were restarted to provide equal environment conditions. Experiments were performed using up to 11 Athlon $2500+$, 512MB RAM machines running Gentoo Linux and JVM $1.4.2$. Obviously, in case of threaded solution only a single machine was used to run the *CIC* infrastructure. Computers were interconnected with a 100 Mbit LAN. The MySQL $4.1.13$ database used by Jena persistence mechanism for storing *yellow pages* data was installed on a separate machine. In all cases the experimental procedure was as follows:

1. Restart of remote agent containers

2. Experiment participants send *ready* message to the *TCA* — just after they are set-up and ready for their tasks

3. On receiving the *ready* signal from *all* agents, the *TCA* sends *start* message to all *QA*s, triggering start of the experiment

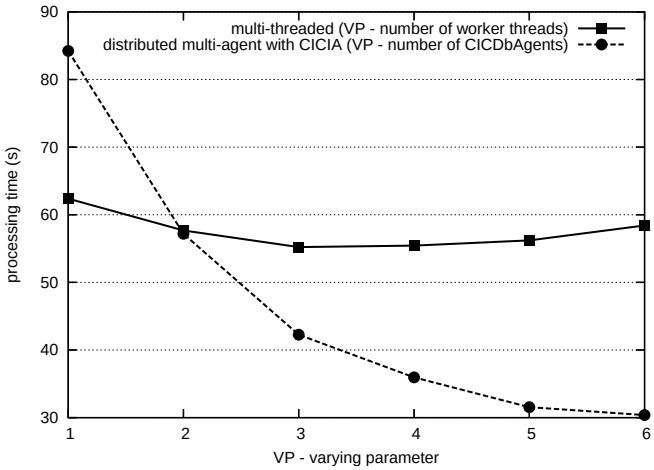

Figure 5: Comparison of threaded and multi-agent architectures; processing time of $10,000$ queries for varying number of worker threads in multi-threaded architecture/number of *CICDbAgents* in multi-agent architectures.

4. When *QA*s receive all results back, they send a ***finish*** message to the coordinator (the *TCA*)

5. Reception of all ***finish*** signals ends the experiment

In Figure 5 we represent the total processing time of $10,000$ requests by the two architectures when the number of agents/worker threads increases from 1 to 6. As it can be seen, the threaded architecture leads to performance gains for up to three threads. While it can be predicted that if multi-core processors were used, further performance gain could have been reported, this approach is limited by the total processing power of the processor. As the number of *CICDbAgents* increases up to 6, the performance of the multi-agent approach improves steadily. We can also observe a leveling-off effect. Therefore, the fact that we were not able to run experiments with more than 11 computers (and thus the largest number of *CICDbAgents* was 6), seems rather inconsequential. Specifically, it can be predicted that if the number of *CICDbAgents* was to increase further, then the performance gain would be only marginal. Overall, with 6 *CICDbAgents* the performance gain over the system with only a single *CICDbAgent* is of order of 3. Furthermore, the performance gain over the implementation based on threads is of order 2 (here, we compare the best threaded solution—with 3 threads—with the multi-agent solution with 6 *CICDbAgents*).

It should be mentioned that we have also evaluated a modification of the above architecture, where the *CICIA* agent becomes both the delegator of requests and receiver of results. This architecture proved to be slightly faster, however it was

more complicated from the conceptual point of view. Therefore, for the sake of design readability we have abandoned that idea as the final solution to our problem. Finally, we have also experimented with few methods for further performance tuning, but the observed gains (reported in [5] were small enough to conclude that the above described approaches can constitute the basis for *CIC* infrastructure implementation.

5 Selecting the team to execute the job

Let us now describe in more details operations involved in selecting a team that is to execute a job. As described above, in Section 3, the first step in this process is for the *User* to provide its *LAgent* with the necessary information such as a job description, negotiation parameters and, possibly, constraints.

5.1 User Input

User provides its *LAgent* with the job description, negotiation parameters and, possibly, execution constraints. In Section 3 we have presented our ontological representation of computational resources. Here, let us focus our attention on the negotiation parameters which are expressed using, what we named, *Grid Yellow Pages Ontology*. Currently, in our work we utilize three negotiation parameters: cost, job start time and job end time. For each of these parameters the user specifies its importance by assigning weight that is later used in the multicriterial analysis (section 5.3). In addition to negotiation parameters we specify execution constraints, e.g., the maximum price that can be charged for the job. Furthermore, we assume that if any of job execution parameters should not be taken into account, then either 0 weight is given or that parameter is not included in the negotiation parameter-set. Alternatively it is also possible that a given parameter is constrained but it is not given a weight. This also shows how ontology-based approach gives us flexible possibilities of expressiveness. The following OWL Lite code snippet represents how these concepts are combined into the negotiation parameter-set.

```
### negotiation parameters ###

:NegotiationSet a owl:Class.

:negotiationParam a owl:ObjectProperty;
        rdfs:domain :NegotiationSet;
        rdfs:range NegotiationParam.

:NegotiationParam a owl:Class.
```

```
:paramWeight
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain: NegotiationParam;
        rdfs:range xsd:float.

:Cost a owl:Class;
        rdfs:subClassOf:NegotiationParam.

:costConstraint
        a  owl:ObjectProperty, owl:FunctionalProperty;
        rdfs:domain:Cost;
        rdfs:range:FloatConstraint.

:costValue
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain:Cost;
        rdfs:range xsd:float.

:JobStartTime a owl:Class;
        rdfs:subClassOf :NegotiationParam.

:jobStartTimeValue
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain:JobStartTime;
        rdfs:range xsd:dateTime.

:jobStartTimeConstraint
        a owl:ObjectProperty, owl:FunctionalProperty;
        rdfs:domain:JobStartTime;
        rdfs:range:TimeConstraint.

:JobEndTime a owl:Class;
        rdfs:subClassOf:NegotiationParam.

:jobEndTimeValue
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain:JobEndTime;
        rdfs:range xsd:dateTime.

:jobEndTimeConstraint
        a owl:ObjectProperty, owl:FunctionalProperty;
        rdfs:domain:JobEndTime;
        rdfs:range:TimeConstraint.

### generic constraints ###

:NegotiationParamConstraint a owl:Class.
```

```
:FloatConstraint a owl:Class;
        rdfs:subClassOf:NegotiationParamConstraint.

:maxFloatValue
        a owl:FunctionalProperty, owl:DatatypeProperty;
        rdfs:domain:FloatConstraint;
        rdfs:range xsd:float.

:minFloatValue
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain:FloatConstraint;
        rdfs:range xsd:float.

:TimeConstraint a owl:Class;
        rdfs:subClassOf:NegotiationParamConstraint.

:minDateValue
        a owl:DatatypeProperty, owl:FunctionalProperty;
        rdfs:domain:TimeConstraint;
        rdfs:range xsd:dateTime.

:maxDateValue
        a owl:FunctionalProperty, owl:DatatypeProperty;
        rdfs:domain:TimeConstraint;
        rdfs:range xsd:dateTime.
```

As shown in the ontology schema, we separated concepts of constraints and parameters—they are defined in separate classes. This allows us to reuse constraints concepts definitions throughout different parameters definitions. For example, DateConstraint is used by JobStartTime and JobEndTime parameters. Currently, via constraints, we can define maximum or minimum value for our parameters; e.g. maximum cost or minimum JobStartTime. Note also that extending this parameter-set by adding, for instance, penalty for not completing job on time, requires only a relatively simple operation of extending our ontology and making minimal changes in agent-codes. This being the case, the focus of our work was on the agent interaction and parameter / constraint utilization, rather than development of a truly realistic parameter-set. Let us now assume that the *User* stated that the cost of the execution is twice as important than the job end time by giving weight 2 to the cost and weight 1 to the job end time. Then the instance of the proposed parameter-set would have the form:

```
@prefix nego:
        <http://gridagents.sourceforge.net/Negotiation#> .

:NegotiationSetInstance a nego:NegotiationSet ;
        nego:negotiationParam [
                a nego:JobEndTime ;
```

```
        nego:paramWeight "1.0"^^xsd:float
] , [
        a nego:Cost ;
        nego:paramWeight "2.0"^^xsd:float
] .
```

In addition to the job-execution related parameter-set, the user specifies the resource describing parameters that are used to query the *CIC* (see section 3 for a detailed description and a sample SPARQL query). As a way for the *User* to communicate input parameters to its *LAgent*, we have implemented a User Agent GUI (see section 6).

Let us now assume that, in response to the query, the *LAgent* obtained from the *CIC* the list of agent teams that have resources necessary to complete the job and has filtered these that are not worthy of its trust, and proceed to describe the *LAgent-LMaster* negotiations.

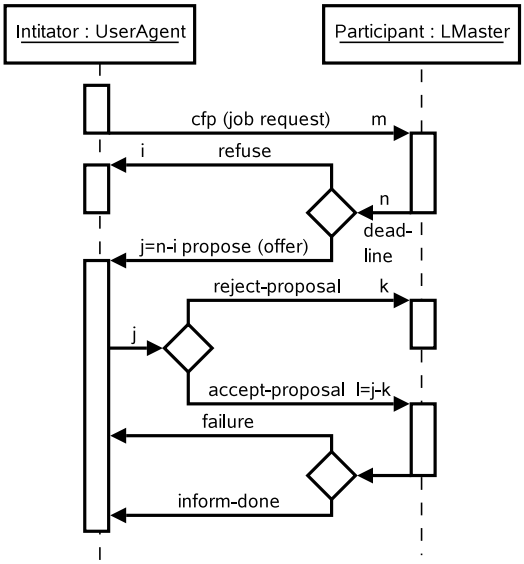

Figure 6: Interaction Diagram of FIPA Contract Net Protocol.

5.2 Negotiations

The *LAgent* utilizes the FIPA Contract Net Protocol to negotiate with *LMaster*s [7]. In Figure 6 we depict a version of the Contract Net Protocol, as it is pertinent to our situation. Note that the *LAgent* negotiates with more than one *LMaster* and therefore the same set of interactions takes place in all of these negotiations.

In the initial step, the *LAgent* sends out the *CALL-FOR-PROPOSAL* (*CFP*) message to all *LMasters* on the final list (after pruning). The *CFP* contains the

job description and the execution constraining parameter-set, according to which *LMaster*s are able to construct their offers (obviously, weights assigned by the *User* to individual parameters are not communicated). In the case of our simplistic parameter-set, the *CFP* message could have the following form:

```
(CFP :sender
        ( agent-identifier
                :name ua@kameleon:1099/JADE
                :addresses (sequence http://kameleon:7778/acc )
                :X-JADE-agent-classname UserAgent )
        :receiver (set ( agent-identifier
                :name lmaster@e-plant:1099/JADE
                :addresses (sequence http://e-plant:7778/acc )) )
        :content  ''((action
          (agent-identifier :name lmaster@e-plant:1099/JADE
                  :addresses (sequence http://e-plant:7778/acc))
                (JobRequest :resRequirements
                (OntoData
                        :ontoDataLang RDF/XML-ABBREV
                        :ontoDataStr
\''<rdf:RDF xmlns:grid="..." xmlns:rdf="...">
        <grid:UnitaryComputer
          rdf:about="jade://request@kameleon:1099/JADE">
        <rdf:type rdf:resource=
        "http://.../Grid#ComputerSystem"/>
        <grid:cpu>
                <grid:cpuClockSpeedMhz
                        rdf:datatype="http://.../XMLSchema#int">
                                1500
                </grid:cpuClockSpeedMhz>
                </grid:cpu>
        </grid:UnitaryComputer>
</rdf:RDF>\''
                                )
                        )
                :negoParamSet
                  (NegotiationSet :negotiationParam
                  (sequence (JobEndTime
                     :jobEndTimeConstraint (TimeConstraint
                     :maxDateValue \"2006-10-12T12:00:00\")))
                                        ))))''
 :reply-with  R1166131532630_0
 :language  fipa-sl0
 :ontology  Messaging
 :protocol  fipa-contract-net
 :conversation-id  C4916061_1166131532629 )
```

In this example the *LAgent* is looking for a single machine with a 1.5 MHz CPU and specifies the deadline for the job execution by constraining the JobEndTime

negotiation parameter.

On the basis of the received *CFP* and their view of the situation on their teams, *LMaster*s prepare their proposals and send them back to the *LAgent*, using *PROPOSE* messages. Note that it is possible that some of *LMasters* refuse the *CFP* (using a *REFUSE* message). For example, in the time between the last update of team information in the *CIC* and the *LAgent*'s request, some resources "disappeared" and the team cannot complete the task. Furthermore, for a variety of reasons (e.g. network congestion) some teams may not respond in time. The positive response message, containing an offer could have the following form:

```
(PROPOSE
  :sender ( agent-identifier
    :name lmaster@e-plant:1099/JADE
    :addresses(sequence http://e-plant:7778/acc)
    :X-JADE-agent-classname LMaster )
  :receiver (set (agent-identifier
    :name ua@kameleon:1099/JADE
    :addresses(sequence http://kameleon:7778/acc)
    :X-JADE-agent-classname UserAgent ) )
 :content ''(
    (result
      (action
        (agent-identifier
          :name lmaster@e-plant:1099/JADE
          :addresses(sequence http://e-plant:7778/acc)
          )
    (JobRequest
            ...
          )
      )
      (JobRequestOffer
        :negoParamSet
          (NegotiationSet :negotiationParam
          (sequence
          (JobStartTime
             :jobStartTimeValue\''2006-10-12T11:30:00\'')
          (JobEndTime
              :jobEndTimeValue\''2006-10-12T12:30:00\'')
            (Cost
              :costValue \''120'')
            )
          )
        )
    )
  )''
  :reply-with ua@kameleon:1099/JADE1166137976099
  :in-reply-to R1166137976093_0
  :language fipa-sl0
```

```
  :ontology Messaging
  :protocol fipa-contract-net
  :conversation-id C4916061_1166137976092
)
```

The response message informs the *LAgent* that the *LMaster* is willing to complete the job and devote resources to it within the specified time-frame and that the total cost will be 120 units. Note that in the Contract Net Protocol, sending a *PROPOSE* message constitutes a *commitment* of the *LMaster* to the conditions it specified.

In the current design of the system the *LAgent* awaits for responses until all of them arrive or a specific deadline occurs. Note that it is necessary to impose a deadline to avoid a deadlock; e.g. it is possible that one of *LMaster*s looses connection to the Internet and cannot communicate back its offer. If after the deadline there is no proposal then the *LAgent* cannot execute the task and reports this fact back to the *User*. Otherwise, if there is at least one offer, the *LAgent* starts evaluating offers. Proposal evaluation is a two-stage process:

- Offers which do not meet execution constraints (e.g. cost, job start time, job end time) are filtered out. If all offers are filtered out at this stage, due to constraints, then the *LAgent* cannot execute the task and reports back to the *User*.
- The remaining offers are evaluated using Multi Criterial Analysis (*MCA*)—see section 5.3.

The first stage of the process requires an explanation. It is reasonable to ask: why would an *LMaster* send an offer that violates constraints that were given to it. This situation is, on the one hand, result of a simplification in our current design of the system; while on the other hand it is a preparation for future system extensions. The simplification concerns the *LAgent*, which filters out all offers that violate constraints. Observe that this may result in very few offers (or no offers at all) left for consideration. This also prevents the *User* from specifying *soft constraints* that represent a "strong preference" but violation of which does not necessarily mean that the offer is unacceptable. For instance, I may prefer to have this job done tonight, but if I can have it done extremely cheap by tomorrow evening, then I may be willing to accept this offer. Therefore, this step constitutes preparation for the future system extension. Its behavior is being prepared for handling exactly such job constraints that are "flexible." In this context it is important to note, that some constraints may actually remain "sharp" and their violation may result in an offer being filtered out; currently we have not decided if the *LMaster* is going to be informed if a given constraint is flexible or not, but we are being swayed toward the solution in which the *LAgent* keeps this information to itself. Now, recall that the *LMaster* has knowledge of the capability of its team and its "pricing

policy" and when it makes an offer, it is going to be one that can be backed up by a Service Level Agreement. Let us assume that an *LMaster* may have a full load for the next 12 hours, but then has no jobs scheduled. In this case it may make an offer which is going to violate the timing constraint—as the execution will start past the suggested deadline—but since it has no tasks scheduled, it may make an extremely cheap offer. We plan to address these types of reasoning and strategizing in the future.

After the *MCA* is applied to the remaining offers, the specific team is selected to execute the job. In this case the *ACCEPT-PROPOSAL* message is sent to the *LMaster* of that team. The remaining teams are rejected by sending to them the *REJECT-PROPOSAL* message. The selected team confirms acceptance by sending back an *INFORM-DONE* message. Obviously, the Contract Net Protocol is also taking care of various "emergency situations;" e.g. failure of the selected team to respond.

5.3 Multi Criteria Analysis

Let us now describe in more detail the Multi Criterial Analysis-based selection process. In the current implementation of the system we use the *linear additive model* [12]. Note that this model was selected for its simplicity. However, it has to be stressed that any other *MCA* method can be applied to evaluate received offers.

In the case of the linear additive model, evaluation is done by multiplying value scores on each criterion by the weight of that criterion, and then adding all those weighted scores together. Recall, that we have three criteria that take part in the MCA process: cost, job start time and job end time. If communication with m teams resulted in n proposals ($m - n$ teams refused, send us proposals that were filtered out, or did not respond within the deadline) then criterion scores of the *i*-th team are calculated as follows:

Start Time Score:

$$STS_i = \frac{\sum_{j=1}^{n}(startTime_j - currentTime)}{(startTime_i - currentTime)}$$

End Time Score:

$$ETS_i = \frac{\sum_{j=1}^{n}(endTime_j - currentTime)}{(endTime_i - currentTime)}$$

Cost Score:

$$CS_i = \left(cost_i \sum_{j=1}^{n} {cost_j}^{-1} \right)^{-1}$$

All scores are normalized and the i-th team final score is calculated as:

$$\text{Team Score } TS_i = startTimeWeight \cdot STS_i + endTimeWeight \cdot ETS_i + costWeight \cdot CS_i$$

Team with the highest overall score, obtained as a weighted sum of individual criterion scores, is selected as the "winner." For the example of the MCA in use, please refer to the next section.

6 Example

Let us now present an example of how our system works to support a *User* who would like to execute a job utilizing the PVM programming library on 16 processing nodes of a single computer. First, our *User* would specify resource requirements using the GUI interface shown in the Figure 7.

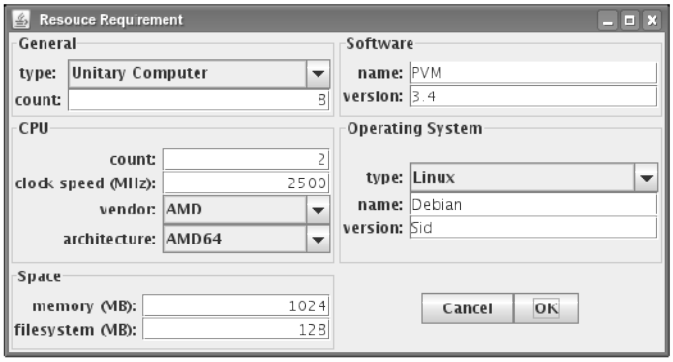

Figure 7: User Agent GUI: Resource requirements.

In the next step, the *User* has to provide its *LAgent* with negotiation parameters expressing her execution preferences. For example, let us assume that she would like to meet the deadline of 11:00 hours on 1st of March, 2007. Furthermore the cost should be no larger that 500 units. Note that the cost does not matter as much as the time—the end time weight is 3, while the cost weight is 2; see Figure 8. Finally, in this example the *User* does not care too much when the job starts (weight 1), she clearly wants to meet a specific execution deadline.

From the specified resource requirements the *LAgent* creates the ontological instance of resource requirements and, based on that instance, constructs a

Figure 8: User Agent GUI: Weights and constraints of criteria.

SPARQL query and sends it (as an ACL request message) to the *CIC* agent. The *CIC* executes the query and as a result delivers a list of candidate teams meeting resource requirements. The answer from the *CIC* contains a list of contacts to *LMasters* representing teams, encapsulated in a *ResultSet*.

Since trust management is not yet implemented, the *LAgent* starts Contract Net Protocol-based negotiations with all *LMasters* found on the list. It receives their offers, filters these that violate constraints and evaluates the remaining ones using the MCA. Figure 9 shows three teams and their scores evaluated by the MCA.

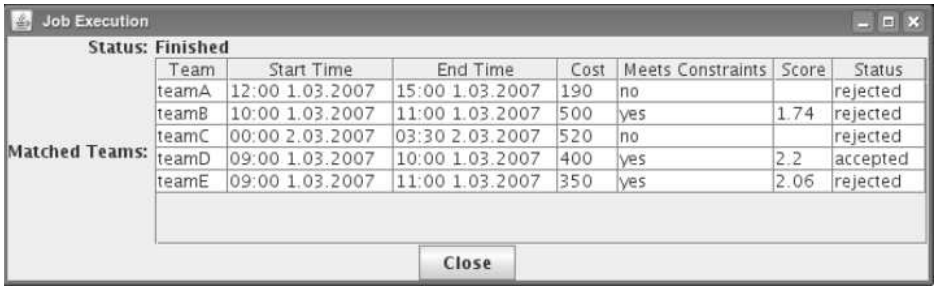

Figure 9: User Agent GUI: Matched teams and their scores.

As we can see, *teamA* and *teamC* have been rejected because they do not meet the deadline constraint. The remaining four teams have been evaluated. Despite of cost proposed by the *teamE* being cheaper by about 30%, it was the *teamD* that was accepted to do a job because of an earlier job completion time. Note that, unless instructed otherwise, the decision is made by the *LAgent* autonomously and

the depiction in Figure 9 is presented only to illustrate the process.

7 Concluding remarks

In this paper we have presented our work devoted to development of an agent-based Grid resource-brokering system. We have focused on three aspects of the proposed system. First, we have discussed conceptual foundations that led to the proposed solution. Second, we have outlined the proposed system. Third, we have summarized our experimental work that lead to a specific way in which we have implemented the *CIC* infrastructure. Finally, we have described and practically illustrated how an agent that wants to select a team to execute its job would act within our system. Our current work is devoted to implementing processes involved in agents selecting team to join. We will report on our progress in subsequent reports.

References

[1] C. Bădică, A. Bădită, M. Ganzha, M. Paprzycki, Developing a Model Agent-based E-commerce System. In: Jie Lu et. al. (eds.) E-Service Intelligence—Methodologies, Technologies and Applications, Springer, Berlin, 2007, 555–578

[2] J. Cao, D. J. Kerbyson, G. R. Nudd, Performance evaluation of an agent-based resource management infrastructure for Grid computing. In: Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, 311–318

[3] K. Chmiel, D. Tomiak, M. Gawinecki, P. Kaczmarek, M. Szymczak, M. Paprzycki, Testing the Efficiency of JADE Agent Platform. In: Proceedings of the ISPDC 2004 Conference, IEEE Computer Society Press, Los Alamitos, CA, 2004, 49–57

[4] M. Dominiak, W. Kuranowski, M. Gawinecki, M. Ganzha, M. Paprzycki, Utilizing agent teams in Grid resource management — preliminary considerations. In: Proceedings of the IEEE J. V. Atanasoff Conference, IEEE CS Press, Los Alamitos, CA, 2006, 46–51

[5] M. Dominiak, W. Kuranowski, M. Gawinecki, M. Ganzha, M. Paprzycki, Efficient Matchmaking in an Agent-based Grid Resource Brokering System, Proceedings of the International Multiconference on Computer Science and Information Technology, PTI Press, 2006, 327–335

[6] O. F. Rana, B. Di Martino, Grid performance and resource management using mobile agents, In: Performance analysis and Grid computing, 2004, 251–263

[7] FIPA Contract Net Interaction Protocol Specification, http://www.fipa.org/specs/fipa00029/SC00029H.html

[8] I. Foster, N. R. Jennings, C. Kesselman, Brain Meets Brawn: Why Grid and Agents Need Each Other, AAMAS'04, July, 2004, ACM Press, 2004, http://www.semanticGrid.org/documents/003-foster_i_Grid.pdf

[9] M. Ganzha, M. Gawinecki, P. Kobzdej, M. Paprzycki, C. Bădică, Towards trust management in an agent-based e-commerce system—initial considerations, In: A. Zgrzywa (ed.) Proceedings of the MISSI 2006 Conference, Wroclaw University of Technlogy Press, Wroclaw, Poland, 225–236

[10] JADE: Java Agent Development Framework. See http://jade.cselt.it

[11] Jena—A Semantic Web Framework for Java. See http://jena.sourceforge.net/

[12] J. Dodgson, M. Spackman, A. Pearman, L. Phillips, DTLR multi-criteria analysis manual, UK: National Economic Research Associates, 2001

[13] S.S. Manvi, M.N. Birje, B. Prasad, An Agent-based Resource Allocation Model for computational Grids, Multiagent and Grid Systems, 1(1), 2005, 17–27

[14] D. Ouelhadj, J. Garibaldi, J. MacLaren, R. Sakellariou, K. Krishnakumar, A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in Grid Computing. In: Peter M. A. Sloot et. al. (eds.), Advances in Grid Computing—EGC 2005, Lecture Notes in Computer Science, 3470, Springer-Verlag, 2005, 651–660

[15] SPARQL Query Language for RDF. See: http://www.w3.org/TR/rdf-sparql-query

[16] H. Tianfield, R. Unland, Towards self-organization in multi-agent systems and Grid computing, Multiagent and Grid Systems, 1(2), 2005, 89–95

[17] D. Trastour, C. Bartolini, C. Preist, Semantic Web Support for the Business-to-Business E-Commerce Lifecycle, Proceedings of the International World Wide Web Conference, ACM Press, New York, USA, 2002, 89–98