Supervising Agent Team in an Agent-based Grid Resource Brokering System—Initial Solution

Wojciech Kuranowski
Software Development Department, Wirtualna Polska
Gdansk, Poland

Maria Ganzha, Marcin Paprzycki
Systems Research Institute Polish Academy of Science
Warsaw, Poland
{maria.ganzha, marcin.paprzycki}@ibspan.waw.pl

Ivan Lirkov
Institute for Parallel Processing, Bulgarian Academy of Science
Sofia, Bulgaria

Abstract

Currently, we are developing an agent-team based infrastructure for resource brokering and management in Grids. In this note we consider how team is supervised and how mirroring can be applied to improve chances of its long-term persistence.

1. Introduction

In our work we follow results presented in [4, 11] and develop a system in which software agents play role of resource brokers and managers in the Grid. To this effect, in [1] we have presented an overview of the proposed approach. In [2] we studied the most effective way of implementing yellow-page based matchmaking services. While in [6] we considered processes involved in agents seeking teams to execute their jobs, and in [9] we described processes taking place when agents seek teams to join. Finally, in [7] we have discussed trust-management-related issues. Since the proposed approach is based on agent teams, here, we discuss how to keep them together and prevent from being dissolved due to "technical difficulties" faced by team leader(s). We start from a brief overview of the proposed approach. Next, we discuss how a team leader controls status of its workers (their availability / existence). In the follow-up section we propose how mirroring can be applied to increase chance of team's survival.

2. Proposed approach

Rationale for the system is based on literature analysis and considering the Grid as an "open environment," (consisting of computers connected to the Internet [4]). A complete set of assumptions and earlier results can be found in [1, 2, 6, 9, 7, 5]. Here, we start by presenting the Use Case diagram of the system (in Figure 1) and discussing most important (for this note) properties of the system.

The *Client Information Center* (*CIC*) agent plays the role of a central repository where information about all other agents is stored. It contains detailed information about teams that look for workers and teams offering to execute a job.Utilization of the *CIC* represents a "yellow page" based approach to matchmaking (see, [12] for critical analysis of possible approaches to matchmaking).

Let us now assume that the system is already running for some time, so that there exist at least some agent teams and their "advertisements" are posted within the *CIC*. The *User* can either try to contribute resources to the Grid or would like to utilize services available there. *User* who wants to contribute resources to the Grid communicates with its agent (the local agent *LAgent* that becomes a worker) and formulates conditions for joining a team. The *LAgent* requests from the *CIC* list of agent teams that satisfy its joining criteria. Upon receiving such a list, due to trust considerations,

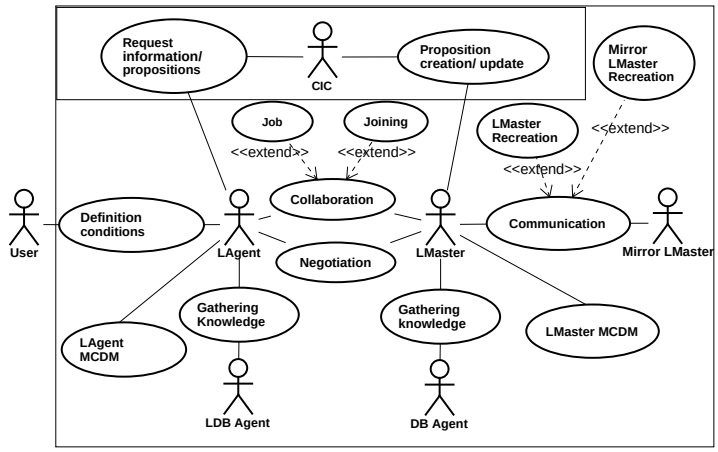

Figure 1. Use Case diagram of the proposed system

it may remove certain teams from the list. For all the remaining teams, the *LAgent* communicates with their *LMasters* utilizing FIPA Contract Net Protocol based negotiations [3] and multicriterial analysis [10] to evaluate obtained proposals. The result of interactions between the *LAgent* and *LMaster*s may be twofold: (1) it finds a team to join, (2) no such team is found (either there were no offers, or they were unacceptable). In this situation the *LAgent* abandons the task and informs its *User*. In the case when the *User* requests that its *LAgent* arranges execution of a task, it specifies conditions of task execution. The *LAgent* queries the *CIC* to find out which teams can do the job. Upon receiving a list of such teams, 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. If no team will satisfy conditions imposed by the *User* the *LAgent* reports this situation and awaits further instructions.

Let us now consider interactions between the *LMaster* and the *LMirror*. Both agents "mirror" each-other's existence and store information necessary to keep the team running, and if the *LMaster* fails, the *LMirror* can take its place. Recall that we assume that the proposed system works in an "open Grid," characterized by potential high volatility of its nodes. In such environment relatively frequent failures of *LMaster*s have to be assumed. However, it has to be stressed that our goal is *not* creation of a "bullet-proof" environment. Rather, we are interested in providing an infrastructure with a reasonable level of resilience against common node failures.

Currently, we assume that the *LMaster* is the "founding father" of the team. Next, when the first agent joins its team, it automatically becomes its *LMirror*. Subsequent agents that join the team are informed by the *LMaster* which agent is the *LMirror*. In this way the *LMirror* becomes a trusted source of information (see below). Furthermore, the *LMaster* registers the *LMirror* with the *CIC*. Here, in the case of a crash of the *LMaster* the *CIC* has to know that a given *LMirror* has a "right" to represent its team and replace the failed agent. The *LMaster* and the *LMirror* share 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 to be executed, knowledge base that stores information about all past interactions with users and workers, etc. Now, it should be obvious why storing such information solely by the *LMaster* would mean that the team would vanish in the case its crash. In order to sustain the team, the *LMaster* and the *LMirror* check each-others existence regularly in short time intervals (see below). In the case when the *LMaster* crashes the *LMirror* takes over its role (becomes the new *LMaster*). The first action of the new *LMaster* is to promote one of worker agents to become its *LMirror* and pass to it all necessary information. Next, it informs all members of the team about its becoming the *LMaster*. Finally, it updates the team information with the *CIC* (informing who is the *LMaster* and the *LMirror*).

Similarly, upon finding that the *LMirror* agent is "gone," the *LMaster* immediately promotes one of workers to become the new *LMirror* and passes to it all necessary information. Next it informs the team as well as the *CIC* who is the new *LMirror*. Note that both the *LMaster* and the *LMirror* may crash "almost simultaneously" and thus the team will be "destroyed."

However, at this stage of the development of the system we consider this outside of scope of our interests.

3. Monitoring status of team workers

As specified above, we assume that nodes in the Grid can disappear at any moment (e.g. a PC being turned off by a playing dog). Therefore, one of key functions of the *LMaster* is monitoring status of it team. The *LMaster* should also be able to evaluate the state of the network between itself and each worker. This latter knowledge can be used, among others, to allocate jobs and to adjust the monitoring procedure (see below). Note that while node crashes should be discovered very fast, the monitoring process should also avoid false positives (i.e. a short network outage should not be misconstrued as a node failure). Thus we have designed a monitoring system based on the *LMaster* "pinging" workers using minimalistic ACL messages. Specifically, each message consists of a string "ping" as its *content*, a *reply-with* field specifying the content of the response (string "re-ping"), and the *reply-by* field specifying the deadline for the response (if the message is obtained after the deadline passed, e.g. due to a network delay, the receiver will ignore it). After receiving a "ping-ACL-message" worker replies to it.Upon receiving a reply, the *LMaster* knows that a given agent is alive. However, to be able to develop a robust, but flexible monitoring subsystem we have designed a somewhat more complicated approach than a simple ping-response. The screen-shot of the GUI of an *LMaster* (in Figure 2) is the basis of its description. We have to stress that this GUI has been used *only* for testing purposes. In actual runs, each *LMaster* obtains (as a text file) a set of parameters and acts autonomously without displaying results in a GUI. The monitoring process consists of the following steps (see Figure 2):

1. *LMaster* performs *X* tests consisting of *Y* pings each (both values *X* and *Y*, as well as other parameters, are configurable and are "an input" to the *LMaster*; we assume that in the future, they will be autonomously adjusted depending on factors like: network conditions, trust etc.).

2. Each ping is send at a certain interval (specified in milliseconds). While we assume that pinging intervals for each agent may differ, currently a ping is send to all agents (message broadcast) and a response is expected within a specified time. The next ping is sent after a specified time has passed.

3. For each agent in the team, a response to a ping is a "pass" if it returns within a required time.

4. For a set of *Y* pings a percent of failed attempts is calculated.

5. After *Y* pings, the "fail-percent" value is compared with the allowed "fail-value" to establish if a given agent has passed one of *X* tests.

6. After *X* tests a total number of failed tests is found.

7. This value is compared with the number of tests that given agent was allowed to fail. Failure of a given number of test may (or may not) result in agent's removal from the team.

8. Counters are zeroed and the procedure is repeated.

In Figure 2 we can see the following values of the above defined parameters of the monitoring system:

- *Ping interval*: time between individual pings is set to 200 ms (the same value used for all agents)
- *Max ping reply*: for a given ping to be scored as a success, the response has to be received within 100 ms (part of the ACL-ping message; see above)
- *Pings per test*: each test consists of 20 pings
- *Number of tests*: each monitoring cycle consists of 15 tests
- *Tests to pass*: for the worker agent to be perceived as alive, it has to pass at least 4 tests (out of 15)
- *Max loss*: a test is passed if the total number of failed responses is smaller than 80% (for 20 pings per test, at least 16 have to be passed)

In Figure 2 we can also see that the *LMaster* is managing 5 agents (bottom left "sub-panel"), while data concerning results of a current liveliness test for the agent *Worker2@home:1099/JADE* is depicted in right top "sub-panel." Specifically, we can see that:

- *Sent* = 18 pings have been send
- *Received* = 18 responses have been received
- *Loss* = 0% losses were recorded (no losses)
- *Min* = 1ms was the fastest response to a ping
- *Avg* = 5ms was the average response time
- *Max* = 41ms was the longest response time
- *Test no.* = 12/15; test 12 (of 15) is in progress
- *Passed* = 11 tests have been passed thus far (all tests, as test 12 is in progress)

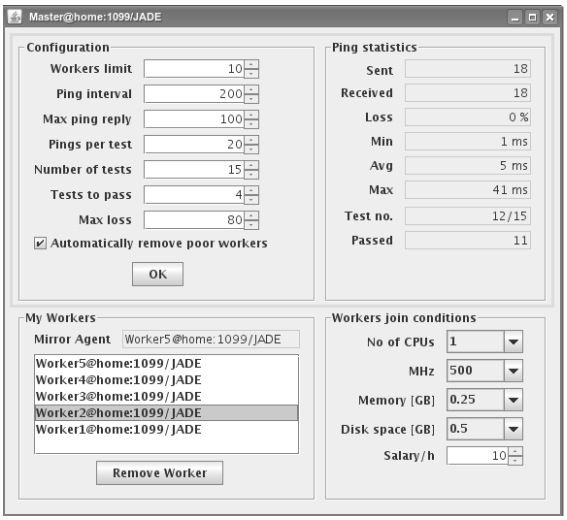

Figure 2. GUI illustrating monitoring of a team member by an *LMaster*

The proposed monitoring procedure is quite flexible. First, let us consider a "local Grid" (e.g. Grid nodes are in the same laboratory). Here, pings can be send often, expected response time can be short, while non-responsive nodes can be immediately removed (see the *Automatically remove poor workers* checkbox). The situation changes on the Internet. First, network outages may (should?) not result in immediate removal of a team member. It is even possible to not to remove any team member, but try to reach it/them, until connection is reestablished (note that this approach can be related directly to trust management, where some nodes are worthy our trust and we try to reach them, while some are not [7]). Furthermore, it is possible to relax selected parameters to avoid unnecessary team member removal. For instance, a number of passed pings within a round of testing can be reduced down to a single passed test (indicating that a node is actually alive).

Note also that testing parameters should be adjusted to the network conditions. For instance, if a packet loss rate is about 70%, then monitoring cannot assume better sucess rate of pinging. Similarly, if the response time is about 100ms, the monitoring procedure has to take this into account. While adaptability of monitoring is not yet implemented, it is easy to see why network conditions may result in testing individualized for each team member. For instance, after each failed test-round probing tests are performed to match the changing network conditions (similarly to the RealPlayer adjusting settings to improve quality of sound).

4. *LMaster–LMirror* interactions

4.1. Re-creating crashed *LMirror* or *LMaster*

Monitoring is used also to find out when either the *LMaster* or the *LMirror* crash and have to be replaced/recreated. Replacing the *LMirror* is relatively simple. The *LMaster* uses monitoring to check liveliness of *all* team members, including the *LMirror*. If the *LMirror* fails the test it has to be recreated. When a team that is working already for some time, the *LMaster* has data about performance (as well as resources) of all workers and can use it to select the best agent to become the next *LMirror*. Obviously, the new *LMirror* should have a "non-stop contract" with the team (see [9]), a very fast network connection and appropriate resources. Promotion of a node to an *LMirror* proceeds in the same way as creation of the first *LMirror*; appropriate modules are send to it to be loaded, followed by all "mirror-data." Finally, the *LMaster* informs the team and the *CIC*, which agent became the new *LMirror*. During this process the *LMaster* stops responding to queries and performing any other tasks, as creation of an *LMirror* has the highest urgency (without its successful completion the team may disappear). Here, we omit the question: what happens if a high-quality node does not exist in the team, as actually not much can be done and such situation is very likely going to, sooner or later, result in team destruction. Furthermore, for the time being we

omit considerations involved in the economical model of being an *LMaster* (i.e. not doing any actual work) and an *LMirror* (which can do some work, but also has to perform "duties" of the mirror, which are of higher priority than completing other tasks). These questions, while very interesting, require a comprehensive solution and are outside of the scope of this note.

When the *LMaster* crashes and the *LMirror* has to become a new *LMaster* we deal with a slightly more complicated situation. First, we need to establish how the *LMirror* finds out that there are problems with the *LMaster*. Obviously, the simplest solution would be if the *LMirror* would listen to ping-tests arriving from the *LMaster* and their disappearance would mean that the *LMaster* has crashed. In this case the *LMirror* could contact the *AMS* agent (FIPA mandated agent that, among others, manages white-page information about agents in the system; *AMS* agent is provided by the JADE agent environment used in our work [8]) to check if the *LMaster* is still registered with the system (the *AMS* agent automatically deregisters defunct agents) and use this information to start the *LMaster* recreation. However, this approach will not work as we assume that the *LMaster* should be able to adaptively adjust ping intervals, (e.g. based on the state of the network). Therefore, the interval between pings may change and the *LMirror* cannot use the "current ping rate" to correctly assess the state of the *LMaster*. Therefore, we have decided to utilize the same monitoring function used by the *LMaster* to monitor team members, for the *LMirror* to monitor state of the *LMaster* (however, the *LMirror* monitors only a single agent). After the *LMaster* stopped responding to test-pings the *LMirror* contacts the *AMS* agent and checks if the *LMaster* is still alive. If it is, it tests its existence again to prevent possible conflicts arising when two agents with the role *LMaster* are created in a single team. If the *AMS* agent confirms that the *LMaster* is gone, the following replacement procedure is initiated:

1. The *LMirror* creates a skeleton agent and loads it with *LMaster* modules. Specifically, the *LMirror* requests that the *AMS* creates an agent based on a class that implements an *LMaster* agent. Here, we assume that it is easier for the *LMirror* to create a new agent, rather than modify itself. Note that there is a large difference between functions performed by the *LMaster* and the *LMirror* agents. For instance, the *LMirror* has only to monitor existence of the *LMaster* and "step-in" when it crashes (separately it executes user-jobs assigned to it by the *LMaster*). At the same time the *LMaster* is responsible for "keeping the team alive," contacts with clients and potential workers, etc.

2. The *LMirror* passes to the new *LMaster* the last version of the team-persisting data and awaits a confirmation that the data has been successfully replicated and assimilated (i.e. it awaits information that the new *LMaster* is fully functional).

3. Upon reception of the required confirmation, the *LMirror* informs all members of the team that a new *LMaster* has been created and will take over management of the team. Note that team members have been informed that the *LMirror* has a "right" to do so, as it can replace the *LMaster* if necessary. Separately, the *LMirror* informs the *CIC* about the change in team leadership (the *CIC* is also ready for such a message to arrive, see above).

4. The current *LMirror* self-destructs as it is no longer needed, while the new *LMaster* takes over. Its first function is re-creation of an *LMirror* (utilizing possessed information about the existing team members). This is the same procedure as when the *LMirror* crashes. When the *LMirror* is fully re-created and data replicated to it, the new *LMaster* informs team members and the *CIC* which agent is the new *LMirror* of the team. Next, the new *LMaster* starts managing the team and interact with incoming messages from both team members and "outside agents." Note that completion of restoration of the *LMaster* + *LMirror* pair is the task that has the highest priority.

5. The new *LMaster* has to pass its current task (recall that the *LMirror* is executing clients' tasks) to a worker, as it will not be able to complete it (the *LMasters* works only on team-management related tasks). Furthermore, it may need to re-allocate the task being executed by the new *LMirror* to another agent (e.g. a high priority task that should not be interrupted by mirroring related activities).

6. Upon reception of a message from the *LMirror*, team members verify that the old *LMaster* is actually no longer available (by contacting it and the *AMS* agent). In the case of the old *LMaster* not responding and the *AMS* confirming that the old *LMaster* is gone, they accept the new *LMaster*. If the old *LMaster* responds to their messages, the message from the *LMirror* is ignored and team members continue to consider the old *LMaster* to be the leader of the team.

4.2. Replication

Thus far we have stated that when the *LMirror* is (re)created and when the *LMaster* is recreated, they

are provided with all data necessary for team management. While this step is relatively obvious, the situation becomes somewhat more complicated afterward. The question is: how often data from the *LMaster* should be passed to the *LMirror* to be replicated. Note that the situation is different than in the case of a local replication (e.g. disk mirroring) as we have to assume that the *LMaster* and the *LMirror* reside on separate machines located in the Internet. One of possible approaches would be to replicate data each time any change occurs within the *LMaster*. This approach, however, is not likely to work for any large and active team. In this case number of information updates within the *LMaster* is likely to be large enough to lock both it and the *LMirror*. Note that each replication requires a confirmation that it was successfully completed. Only after such confirmation the *LMaster* would be able to continue its work. This mechanism is similar to what happens in the case of a standard database information replication procedure (with the difference that the database is usually located within a single data center).

The second possibility would be to send data to the *LMirror* and not wait for a confirmation. The danger is that some data will be lost (and we will have no control over what is actually replicated). In this way, over time the *LMirror* will contain only "random" data, which will not lead to correct re-creation of the *LMaster*.

Finally, it is possible to send updates in packages in specific time-intervals. When such a package is sent, the *LMaster* awaits confirmation of successful replication. Obviously, as a result of such procedure, re-creation of the *LMaster* may result in creation of its somewhat outdated version (crash will happen sometime between replications), but at least data will be consistent up to the time of the last update. Therefore we assume that this approach is the best possible.

5. Concluding remarks

In this note we have presented our work devoted to development of an agent-based Grid resource-brokering system. We have focused on issues involved in team management. First, we have introduced the team-agent-status monitoring tool, which we have implemented. This tool, being highly parameterized can be used as a centerpiece for adaptive team monitoring approach, which we are currently developing. We have also discussed processes involved in interactions between *LMaster* and *LMirror* agents, which are crucial for the team long-term survival. Across the note we have indicated a number of research directions that we plan to pursue. Another one of them, which was not mentioned thus far, is the necessary experimental work needed to establish the actual overhead of the proposed approach, especially in the case of a large team (note that ACL messaging introduces a number of overheads). We will report on our progress in subsequent reports.

References

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

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

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

[4] 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`

[5] M. Dominiak, M. Ganzha, M. Gawinecki, W. Kuranowski, M. Paprzycki, S. Margenov, I. Lirkov, Utilizing Agent Teams in Grid Resource Brokering, International Transactions on Systems Science and Applications,2007, in press

[6] M. Dominiak, M. Ganzha, M. Paprzycki, Selecting grid-agent-team to execute user-job—initial solution, Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems, IEEE CS Press, Los Alamitos, CA, 2007, 249–256

[7] M. Ganzha, M. Paprzycki, I. Lirkov, Trust Management in an Agent-based Grid Resource Brokering System—Preliminary Considerations. In: M. Todorov (ed.), Applications of Mathematics in Engineering and Economics'33, American Institute of Physics, College Park, MD, 2007, 35–46

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

[9] W. Kuranowski, M. Paprzycki, M. Ganzha, M. Gawinecki, I. Lirkov, S. Margenov (2007) Agents as resource brokers in grids—forming agent teams. In: Proceedings of the LSSC Meeting, Springer (to appear)

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

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

[12] 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