Combining an JADE-agent-based Grid infrastructure with the Globus middleware—Initial Solution

Mehrdad Senobari
Tabriat Modares University Tehran, Iran
senobari@modares.ac.ir

Michal Drozdowicz, Marcin Paprzycki
Wojciech Kuranowski, Maria Ganzha,
Systems Research Institute PAS,
Warsaw, Poland
marcin.paprzycki@ibspan.waw.pl

Richard Olejnik
University of Sciences and Technologies
of Lille, Lille, France

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

Abstract

Currently, we are developing an agent based infrastructure for resource management in Grids. In the past our attention was focused on high-level processes involved in agents selecting a team to join or a team to execute a job. In this note we consider how the proposed agent-based system can interact with an actual Grid middleware. As our initial target we have selected the Globus middleware. Here, we present a simple way of submitting a job and receiving results and discuss implementation details.

1 Introduction

Since 1999 we are told that Grid computing will provide a new way to utilize heterogeneous, geographically distributed, multi-domain computer resources [17]. Unfortunately, the uptake of the Grid, while speeding-up recently, is still unsatisfactory. Some important issues that need to be solved are: (a) complicated resource brokering and management in existing Grid middlewares, (b) lack of interoperability between individual middlewares, and (c) too high expectations put on potential users of the Grid (they have to know / learn too much to be able to use it effectively, especially in the early stages of Grid adoption). In this context, it was suggested that software agents combined with ontologies may provide the necessary infrastructure, by infusing Grid with "intelligence." Thus, we follow guidelines put forward in [16, 24]. While arguments presented there are not uncontroversial we accept them as a starting point, and propose an approach based on agents working in teams. Specifically, in [13] we presented an initial overview of the proposed approach. In [12] we followed with a study ways of implementing yellow-page based matchmaking services. While in [11] we considered processes involved in agents seeking teams to execute their jobs, in [21] we have discussed processes taking place when agents seek teams to join, and in [20] we discussed how team is kept together through mirroring and presence monitoring. Separately, in [19] we have discussed trust-management-related issues. Finally, in [14] we have contemplated how the proposed approach can be intermixed with work done within the *ADAJ* project (see, also [23]).

Except for the last paper, our work was focused on high-level (intelligent resource brokering and management) functions of the system. Overall, we have assumed that at a certain moment a job is going to be passed from the "agent system" to the "work environment", where it is going to be executed. Next the results will be returned to the agent system and delivered to the user by her/his representative agent. How this process can be implemented is the focus of this note. Obviously, a job could be executed on a single "home PC" by an agent that represents it. However, what is more interesting is the question of how a job can be passed to an actual Grid infrastructure. Therefore, we have selected Globus, as one of the most popular Grid middlewares ([15]), and established how an agent can pass a job to the Globus infrastructure to be executed and then receive the results back to be delivered to the user.

2 Overview of the proposed approach

Let us start from a brief description of the proposed utilization of software agents in Grids. The main assumption behind our system is that we consider the Grid as an "open

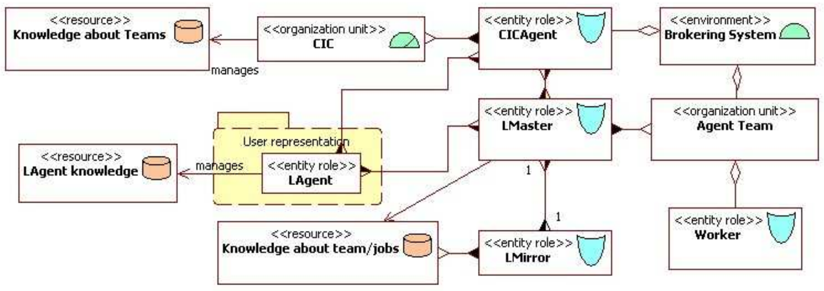

Figure 1. AML social diagram of the proposed system

environment" (consisting of computers connected to the Internet [17]), rather than a "local grid" (consisting of machines in a closely monitored and managed environment). To discuss main actors and processes in the system, in Figure 1 we present the AML social diagram ([8]), which complements the UML use case diagram found in [10]).

In general, we have to consider two key functions of the system: (a) helping the user to contribute its resources to the Grid (and be paid for doing so), and (b) helping the user to execute the job within the Grid. Let us start with the first case (more details can be found in [21]). Each user is represented in the system by an agent (the *LAgent*). This agent provides an intelligent interface between the user and the *Brokering System* and in this way can be viewed as a *Personal Agent* (following the concept proposed in [22]). After the user specifies conditions of joining an agent team, the *LAgent* contacts the *Client Information Center* (*CIC*); represented in the system by a *CICAgent*) to obtain a list of teams it can potentially join. The *CIC* is a central repository of *Knowledge about Teams*. Specifically, it contains information about teams that look for workers (with specification of their expected characteristics), teams offering to execute a job (including specification of available resources), and individual agents registered with the system. Note that we assume that while all agents within a platform are known to it, only agents registered with the *CIC* can use services of our system.Utilization of the *CIC* represents a "yellow page" based approach to matchmaking (see, [25] for critical analysis of possible approaches to implementing matchmaking in a distributed system).

Upon receiving a list of teams that can be interested in its offers, the *LAgent* may remove some of them due to trust considerations [19] (utilizing the *LAgent knowledge*. Next it utilizes the FIPA Contract Net Protocol based negotiations ([1]) and multicriterial analysis [9] to decide which team to join. Negotiations involve the *LAgent* and the *LMaster*s representing their teams. *LMaster*s utilize the *Knowledge about team/jobs* resource to prepare their offer ([11]). The result of negotiations can be twofold: (1) the *LAgent* finds a team to join, (2) no such team is found (in this case the *LAgent* informs the user about it, and awaits further instructions, which may involve modification of constraints, or an instruction to abandon the task completely). Upon joining the organizational unit (*Agent Team*), the *LAgent* is appropriately configured to undertake the role of a *Worker*.

Let us now concentrate on the second scenario—search for a team to execute a task—as it provides the basis for the remaining parts of this note. Here, the user communicates to its *LAgent* conditions of task execution (e.g. necessary hardware requirements, or maximal price). The *LAgent* queries the *CIC* to find which teams can execute it task. Upon receiving a list of such teams, the *LAgent* removes from it teams that cannot be trusted [19]. Next, it communicates with *LMaster*s of the remaining teams and utilizes FIPA Contract Net Protocol and multicriterial analysis to find the best one to execute its job. *LMaster*s utilize the *Knowledge about team/jobs* resource to prepare their offers ([21]). As in the previous scenario, if no team satisfies imposed conditions the *LAgent* reports this situation to the user and awaits further instructions. Note that in both cases the decision making process implemented in the system is fairly rudimentary, but it can be easily extended to involve a large space of factors influencing the final outcome.

Finally, in Figure 1 we can see the *LMirror* agent, which "mirrors"/"duplicates" the *LMaster*. Here, both agents store information necessary to keep the team running. In the case of failure of the *LMaster* (*LMirror*), the *LMirror* takes its place (the *LMaster* "promotes" one of its *Workers* to be a new *LMirror*); see [20] for more details.

3 Combining agents and Globus—preliminary considerations

Let us now focus on in interfacing the above described agent-based management infrastructure with an actual computational infrastructure. One of overarching design guidelines for the system is an attempt to hide complexities of the underlying environment from the end-user. For instance, it is the *Worker* agent that has to know how to utilize the computational infrastructure it represents. Thus, the user should be asked mainly to point to files that the job consists of and resources it utilizes (e.g. a binary executable file ready to run on an Intel x86 processor and a data file). The remaining actions should be undertaken autonomously "by the system." Taking into account assumed capabilities of software agents ([27]), it is easy to see that to be able to interface with multiple computational infrastructures it is enough to have a *Worker* agent devoted to each one of them. For example, one *Worker* may know how to run a job on a "home PC," while another *Worker* may know how to run it within the Globus-based Grid. Note that at this stage we *do not* attempt to completely bridge the heterogeneity gap and be able to run a job, for instance, partially within the Unicore, and partially within the EGEE middlewares, while finishing it on a Cell-processor-based supercomputer. Thus far we consider only a scenario when a single job is executed within a single "infrastructure" represented by a single *Worker* agent.

Taking this into account, the *Worker* agent should be in part independent of the runtime environment. To achieve this goal we propose that the *Worker* be comprised of two parts. First, the agent system specific modules that allow it to communicate with the user and the *LMaster*, receive messages from the *LMirror*, etc. These modules are to be the generic for all *Worker* agents in the system. Second, the "computational-infrastructure-specific" functionalities, that should be encapsulated into a separate module, content of which depends on the infrastructure that the given *Worker* is to represent. In this way, the infrastructure-specific "part" of the *Worker* allows utilization of various middlewares (or even running the job directly on a PC/workstation it represents); depending on the way it has been configured. Moving in this direction, we have designed a *Job Executor* module, which provides an abstract interface to execute the user job. Note that in the system under development, we plan to utilize the modular agent design introduced originally by T. Tu and collaborators in the DynamiCS project (see, [26] for more details). There, it was shown, how agents can be composed on demand from specified components. More recently, in [18] we have described how this idea can be implemented in a JADE-based agent environment ([6]). Therefore, one should envision, that when the *LAgent* becomes a *Worker*, then an appropriate *Job Executor* module is loaded and it allows the resulting *Worker* agent to interact with the specified computational infrastructure.

Let us now consider functionalities that are needed within the *Job Executor* module in the case of executing jobs within the Globus Toolkint 4 environment. Here, after consulting the literature, we have considered two different approaches:

- In the first approach, we simply used the standard Globus Toolkint 4 job submission client, the GRAM4 *globusrun-ws* ([5]). In this case a special instance of the *Job Executor* module calls GRAM4 commands from within the *Worker* agent. The only requirement is to provide the proper argument string for the job submission command. The main advantage of this solution is its simplicity. No knowledge of the internal GT4 operations, except of basics of commands of the GRAM4, is needed for job submission. The disadvantage is that monitoring the job status requires "manual" checking and interpreting output of the command. Furthermore, the most natural implementation of this solution is OS-dependent. Specifically, this solution may directly utilize the `Runtime.exec()` method to run the client command. However, as one can find (e.g. see in [7]), this method "is not cross-platform." Thus, for instance, in Windows parameters are different than in Linux. In turn, this would mean that we would have to generate a separate *Job Executor* module for each operating system that requires a unique parameter string. Such solution, while feasible, is definitely not appealing. Note that since Java 1.6, a newer utility named `ProcessBuilder` was added to the JDK. It provides some new features that `Runtime.exec()` does not have. However, both of them are similar since they are not shell-command processors, they are only a way to start a new process. Thus even utilization of the `ProcessBuilder` would require writing separate modules for different OS's and thus its usability is also restricted in the needed context.

- The second approach utilizes the GT4 *Java WS Core* [4]. In this approach we retain direct control over the job execution, as well as monitoring its status. This solution exploits capabilities offered by the GT4 `Java WS Core`, thus provides an OS-independent implementation that runs on all platforms that the GT4 can potentially run on. The *GramJob* API [2] within the *Java WS Core* provides all the necessary methods to submit a job using GRAM4 and control its lifetime. The only "disadvantage" of this approach is that the implementation is somewhat complicated (vis-a-vis the first solution). However, much of the needed code has been already written by the Globus team.

It should be stressed that the decision to use the GT4 *Java WS Core* provides us also with an easier control over job execution, which is needed not only on the level of the *Worker* agent (it needs to know what is happening with jobs submitted to the Grid it represents), but also on the level of the *Agent Team*. In the latter case, the *LMaster* has to be able to know what is happening with jobs being executed by its workers to be able to act proactively to assure fulfillment of the *Service Level Agreement*. Obviously, both approaches described above provide us with some possibility of controlling the status of job execution. However, in the first, to find out the job status one has to call the `globusrun-ws` client, and then interpret the output of the command. In the case of the *Java WS Core*, information about the job status is provided by the environment (push approach). Therefore, also from this perspective, the *Java WS Core*-based approach seems to be more appropriate.

4 Combining agents and Globus—implementation details

Job Executor is one of the key modules of the *Worker* agent. It provides an abstract interface to execute the user job within the infrastructure that the *Worker* represents. We have designed two implementation of the *Job Executor*. First one runs the job as a normal process within the worker machine (*Simple Job Executor*). Note that even in this simple case it is also possible that the *Worker* represents multiple machines. Here, the technology provided by the JADE agent environment can be used. In this case, the JADE agent platform encompasses multiple (possibly heterogeneous) machines. It is the *Worker* that decides which machine should be used to actually execute a given job.

The approach allows to run the job in a Globus node (*Globus Job Executor*). This *Job Executor* enables the *Worker* agent to submit a job to the local Grid resource manager using the GRAM4 *Java WS Core*, and track its lifetime. Note that selection of the proper *Job Executor* is done when the *Worker* agent is configured to fulfill this role, and depends on the infrastructure that it is supposed to represent. Here, the *LAgent*, before becoming a *Worker* does not (need to) have the *Job Executor* module installed. However, to be able to manage the process of joining a team it has to know what the computational infrastructure it represents is. This is needed first, to be able to query the *CIC*, and second, to negotiate with *LMaster*s of selected teams. Furthermore, if the process of joining a team is unsuccessful there is no need for this module to be installed (there is no work coming). Instead, acting on orders of its user, the *LAgent* may become an *LMaster* of a new team and need a completely different set of modules to fulfill this role.

4.1 Job execution scenario

Let us now assume that (a) a given *LAgent* has identified a specific team that is to execute its job, and (b) that the selected team is a front end to the Globus Grid middleware. In this case the sequence of actions leading to executing user request is as follows.

- The *LAgent* communicates with the *LMaster* of the selected team and sends the job executable and its necessary parameters. This is done by an ACL message which contains an instance of the *JobExecutionAction* in its action slot (all ACL messages exchanged between agents are carried by the JADE message transport subsystem):

```
(REQUEST
:receiver
  (set
   ( agent-identifier
       :name masterA@MS-JADE
       :addresses (sequence
          http://192.168.242.66:7778/acc )) )
       :content
          ``((action
           (agent-identifier
              :name masterA@MS-JADE
              :addresses (sequence
                 http://192.168.242.66:7778/acc))
           (JobExecutionAction
              :ExecutionParams
          (JobExecutionParams
              :Executable <binary representation
                              of the job>
              :Arguments \``-stage 1200\'')) )
          )''
 :language   fipa-sl0
 :ontology   messaging
 :protocol   fipa-request)
```

As the *LAgent* does not know which *Worker* agent will fulfill the request, it sets the selected *LMaster* as the actor of the *JobExecutionAction*. Another parameter of this message is an instance of the *JobExecutionAction*, which has two inner arguments: *Executable* and *Arguments*. The *Executable* contains the binary representation of the job, and *Arguments* are the necessary arguments to run that job. Note that here we present an example in which the job consists of an executable and its parameters. This is also what has been implemented. However, this approach generalizes naturally to sending other forms of requests, e.g. source codes and input data. We plan to extend functionality of the system in this direction next.

- The *LMaster* selects one of its *Worker* agents and requests it to perform the job (criteria of selection can be based, among others, on resource characteristics and trust considerations, but this is out of scope of this note). This is done by forwarding the content of the above listed ACL message to the selected *Worker*.

- Upon receiving the message, the *Worker* agent uses the *Globus Job Executor* module to submit the job into the Globus infrastructure. For this, it first extracts and saves the job executable in its local directory and then creates a job description, which is an XML file based on the GRAM4 job description schema [3]; that could look as follows:

```
<job>
 <executable>
          /opt/aig/shared/[jobID]/Job@userA
 </executable>
 <argument>—stage 1200</argument>
 <stdout>
          /opt/aig/shared/[jobID]/output
 </stdout>
 <stderr>
          /opt/aig/shared/[jobID]/error
 </stderr>
</job>
```

Here, we can see that the job is assigned a unique ID (*jobID*). Furthermore, specific directories are created to save its outputs and, with the help of the `stdout` and the `stderr` descriptors, outputs are redirected to separate files.

The above job description is passed to the *Globus Job Executor*, which in turn uses the *GramJob* utility of the GT4 *Java WS Core* to submit a new job using the *WS-GRAM4* and then listens to changes in its state. The *GramJob* facilitates submitting a job, canceling it, sending a signal command and registering and unregistering job state change listeners. Job submission can be done in the following way:

```
File rslFile = new File(path to the xml file);
GramJob job = new GramJob(rslFile);
job.addListener(listenerInstance);
job.submit(factoryEndpoint,
          false, true, submissionID);
```

Here, the `GramJob` class needs an instance of Java `File` that points to the path of the job description file. The `addListener` method of the `GramJob`, accepts an object that implements the `GramJobListener`. Finally, the job is to be submitted using the `submit` method, which needs the URL of the GRAM4 web-service, and the submission ID (*jobID*, above) of the job.

The GRAM4 can interact with many Grid *Local Resource Managers* (*LRM*) like LSF, PBS, Condor, or can directly use the fork mechanism provided in UNIX. It is therefore possible to select the desired *LRM* when submitting the job (using the `job.submit()` method), or leave it to the GRAM4 to select the default one. In the sample job description above we let the GRAM4 to select the default *LRM*. Overall, the primary parameter that the *Worker* agent should "know" to be able to submit a job to the Globus is the URL of the GRAM4 service. Specifically, the *factoryEndpoint* in the above snippet is an instance of the *EndpointReferenceType*, which is used to point to the GRAM4 web-service. The default contact string is configured at the *Worker* configuration. The default one is:

```
https://localhost:8443/wsrf/services/
                    ManagedJobFactoryService
```

which is the address of *ManagedExecutionJobService*, a service running within the Globus service container.

- Upon job termination, which corresponds to one of *DONE* or *FAILED* states in the `GramJob`, the result of execution, and its output(s) (if any) have to be sent back to the corresponding *LAgent* ("owner" of that job). Note that the *Worker* does not know the *LAgent*. Thus all the information has to be passed through the *LMaster*. Obviously, in the case of a large team working on relatively "small/short" jobs, this may generate a communicational bottleneck within the *LMaster*. As a result the *LMaster* may not be able to properly manage the team; e.g. respond to queries concerning possibility of execution of further jobs. This is one of the research issues that will be addressed both theoretically and experimentally in the near future.

  As one can see in the sample job description file (above), the output of the job could be redirected to one or more separate files. Note that the *Worker* agent knows the content of the job description as it is its creator. Therefore, upon receiving the *DONE* or *FAILED* information from the `GramJob`, it can collect the resulting data, from where they have been send to, and pack everything into an ACL message. This message is then sent to the *LMaster*, which forwards it to the *LAgent* that submitted the job; thus completing the process.

5 Concluding remarks

In this note we have considered how an agent infrastructure designed to provide "the brain" for the Grid, can connect with "the brawn" to execute user requests within it. For the case study we have selected the Globus Grid middleware (GT4). We have described in some detail how we have selected the solution and how it is to work. The minimalistic version of the approach (dealing only with execution of binary files) has been implemented and the code can be found within the Sourceforge repository. Currently we are proceeding to make the implementation more robust and flexible and will report on our progress in subsequent publications.

Acknowledgment

Work of the Polish team was in part 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 N N516 382434). Collaboration of the

Polish and Bulgarian teams is partially supported by the *Parallel and Distributed Computing Practices* grant. Collaboration of Polish and French teams is partially supported by the PICS grant *New Methods for Balancing Loads and Scheduling Jobs in the Grid and Dedicated Systems*.

References

[1] Fipa contract net protocol specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`.

[2] Globus toolkit 4.2.0 javadocs. `http://www.globus.org/api/javadoc-4.2.0/`.

[3] Gt 4.2 gram4: Job description schema document. `http://www.globus.org/toolkit/docs/4.2/4.2.0/execution/gram4/schemas/gram_job_description.html`.

[4] Gt 4.0: Java ws core. `http://www.globus.org/toolkit/docs/4.2/4.2.0/common/javawscore/`, 2008.

[5] Gt 4.2.0 gram4: Developer's guide. `http://www.globus.org/toolkit/docs/4.2/4.2.0/execution/gram4/developer/gram4DeveloperGuide.pdf`, 2008.

[6] Jade—java agent development framework. `http://jade.tilab.com/`, 2008.

[7] When runtime.exec() wont. `http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html?page=3`, 2008.

[8] R. Cervenka and I. Trencansky. *Agent Modeling Language (AML): A Comprehensive Approach to Modeling MAS*. Whitestein Series in Software Agent Technologies and Autonomic Computing. A Birkhauser book, 2007.

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

[10] M. Dominiak, M. Ganzha, M. Gawinecki, W. Kuranowski, M. Paprzycki, S. Margenov, and I. Lirkov. Utilizing agent teams in grid resource brokering. *International Transactions on Systems Science and Applications*, 3(4):296–306, 2008.

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

[12] M. Dominiak, W. Kuranowski, M. Gawinecki, M. Ganzha, and M. Paprzycki. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 327–335. PTI Press, 2006.

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

[14] M. Drozdowicz, M. Ganzha, W. Kuranowski, M. Paprzycki, I. Alshabani, R. Olejnik, and M. Taifour. Software agents in *ADAJ*: Load balancing in a distributed environment. In M. Todorov, editor, *Applications of Mathematics in Engineering and Economics'34*, 2008. to appear.

[15] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[16] I. Foster, N. R. Jennings, and C. Kesselman. Brain meets brawn: Why grid and agents need each other. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, Washington, DC, USA, 2004. IEEE Computer Society.

[17] I. Foster and C. Kesselman. The grid: Blueprint for a new computing infrastructure. 1999.

[18] M. Ganzha, M. Gawinecki, M. Szymczak, G. Frackowiak, M. Paprzycki, M.-W. Park, Y.-S. Han, and Y. Sohn. Generic framework for agent adaptability and utilization in a virtual organization—preliminary considerations. In J. Cordeiro et al., editors, *Proceedings of the 2008 WEBIST conference*, pages IS–17–IS–25. INSTICC Press, 2008. to appear.

[19] M. Ganzha, M. Paprzycki, and I. Lirkov. Trust management in an agent-based grid resource brokering system—preliminary considerations. *Applications of Mathematics in Engineering and Economics'33*, pages 35–46, 2007. M. Todorov (ed.), American Institute of Physics, College Park, MD.

[20] W. Kuranowski, M. Ganzha, M. Paprzycki, and I. Lirkov. Supervising agent team an agent-based grid resource brokering system—initial 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.

[21] W. Kuranowski, M. Paprzycki, M. Ganzha, M. Gawinecki, I. Lirkov, and S. Margenov. Agents as resource brokers in grids—forming agent teams. In *Proceedings of the LSSC Meeting*, LNCS. Springer, 2007.

[22] P. Maes. Agents that reduce work and information overload. *Commun. ACM*, 37(7):30–40, 1994.

[23] R. Olejnik, F. Fortis, and B. Toursel. Webservices oriented datamining in knowledge architecture. *Future Generation Computer System*. to appear.

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

[25] D. Trastour, C. Bartolini, and C. Preist. Semantic web support for the business-to-business e-commerce lifecycle. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 89–98, New York, NY, USA, 2002. ACM Press.

[26] M. Tu, F. Griffel, M. Merz, and W. Lamersdorf. A plug-in architecture providing dynamic negotiation capabilities for mobile agents. In K. Rothermel and F. Hohl, editors, *Proceedings MA'98: Mobile Agents*, volume 1477 of *LNCS*, pages 222–236. Springer-Verlag, 1999.

[27] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.