Software Agents in *ADAJ*: Load Balancing in a Distributed Environment

Michal Drozdowicz, Maria Ganzha, Wojciech Kuranowski, Marcin Paprzycki
Systems Research Institute,
Polish Academy of Sciences, Warsaw, Poland
email: maria.ganzha@ibspan.waw.pl

Iyad Alshabani, Richard Olejnik, Mahmoud Taifour
Computer Science Laboratory of Lille (UMR CNRS 8022),
University of Sciences and Technologies of Lille, Lille, France

Mehrdad Senobari
Department of Computer Science,
Tarbiat Modares University, Tehran, Iran

Ivan Lirkov
Institute for Parallel Processing,
Bulgarian Academy of Science

Abstract

Adaptive Distributed Applications in Java (*ADAJ*) is a platform developed for execution of distributed applications in Java. The objectives of this platform is to facilitate application design and to efficiently use the power of distributed computing. The *ADAJ* offers both a programming and an execution environment. In the latter it implements object observation and load balancing mechanisms. The observation mechanism allows estimating of the JVM load for each node running the *ADAJ* client. The load balancing mechanism dynamically adapts the workload across the system according to this information. Here we discuss how the original design based on JavaParty is going to be superseded by utilization of software agents.

Keywords: distributed applications, adaptive load balancing, software agents

1 Introduction

The *Adaptive Distributed Applications in Java* (*ADAJ*) is a platform developed for efficient implementation and execution of distributed applications in Java. The *ADAJ* offers both a programming and an execution environment. To facilitate efficiency of code execution the *ADAJ* implements object observation which allows estimation of the JVM load on each of its nodes. This information, in turn, allows for utilization of load balancing mechanisms, which can dynamically adapt execution of applications to equalize loads between computers in the system.

The original *ADAJ* was implemented utilizing the JavaParty ([5, 29]) and Java/RMI platforms according to a multi-layer structure using several APIs. Here, the JavaParty provided an execution environment for running distributed applications as well as a mechanism for object migration. The main advantages of utilization of JavaParty were:

- object tracking done by the platform,
- transparency of object placement and execution,

- high efficiency of the solution.

However, it had also some serious disadvantages:

- extremely tight integration with the lower layer of RMI communication,
- utilization of proprietary protocols (KRMI),
- problems of integration with component/service oriented frameworks ([7, 9]),
- unknown future direction of development and release cycle timing.

Furthermore, the original design of *ADAJ* had the following drawbacks:

- was not service oriented and thus did not adhere to the vision of *Service Oriented Computing*,
- *ADAJ* applications had to be written utilizing the JavaParty, so the application had to be tightly coupled with the platform,
- did not provide architectural view, but multi-layer view of the platform (allowed for separation of layers but no separation of concerns).

On the basis of these premises we are in the process of re-designing of the *ADAJ* platform. The main goals for *ADAJ 2* are:

1. create *ADAJ* based on *Service Oriented Architecture* principles,
2. remove JavaParty and replace it with a more flexible brokering mechanism,
3. integrating a dynamic deployment model for the *ADAJ* service oriented applications.

In this paper, we discuss how results put forward in the *Agents in Grid* project (see [18, 28, 19, 17, 21, 16, 22, 23, 24], and software agents in particular, can be utilized on a lower level (within the new *ADAJ*) to:

- detect load imbalance,
- facilitate migration of objects from heavily loaded machine(s) to lightly loaded ones.

To this effect we proceed as follows. In the next two sections we briefly describe the original *ADAJ* project and the *AiG* project. We follow with an outline of a solution that would combine the two project on a high level, where the *AiG* would provide user interface and resource management capability for the *ADAJ*, which would become the work handling infrastructure. Next we discuss how software agents can be introduced into the *ADAJ* to handle load observation and balancing.

2 *Adaptive Distributed Applications in Java* project overview

Let us start from a brief overview of the original ADAJ project(see [26, 8, 27].

2.1 General *ADAJ* architecture

The original *Adaptive Distributed Applications in Java* (*ADAJ*) was a programming and execution system for distributed applications. The *ADAJ* provided distributed collections and the mechanism of asynchronous invocations. In addition, *ADAJ* carried out load balancing in order to improve the performance of executed applications. This mechanism of load balancing was based on application of object redistribution. Its mechanisms were based on exploiting and combining information from the load observation system of the execution platform and from the observation system of dynamic relations between objects of applications running in *ADAJ*. The original *ADAJ* was implemented utilizing the JavaParty and Java/RMI platforms according to a multi-layer structure using several API's. Its main components, represented in figure 1 were:

- Java Virtual Machine (JVM) was regarded as a homogeneous base for construction of distributed applications.
- Remote Method Invocation (RMI, [3]) allowed objects located within various JVM's to communicate between each other as if they were located on the same JVM (using a mechanism of stub/skeleton).
- JavaParty provided an environment of execution of applications distributed on a collection of workstations connected by a network. JavaParty extends the Java language to make it possible to express relatively transparent distribution as well as provides a mechanism for object migration.
- CCA component framework provided services for creating and using CCA compliant components in the platform.

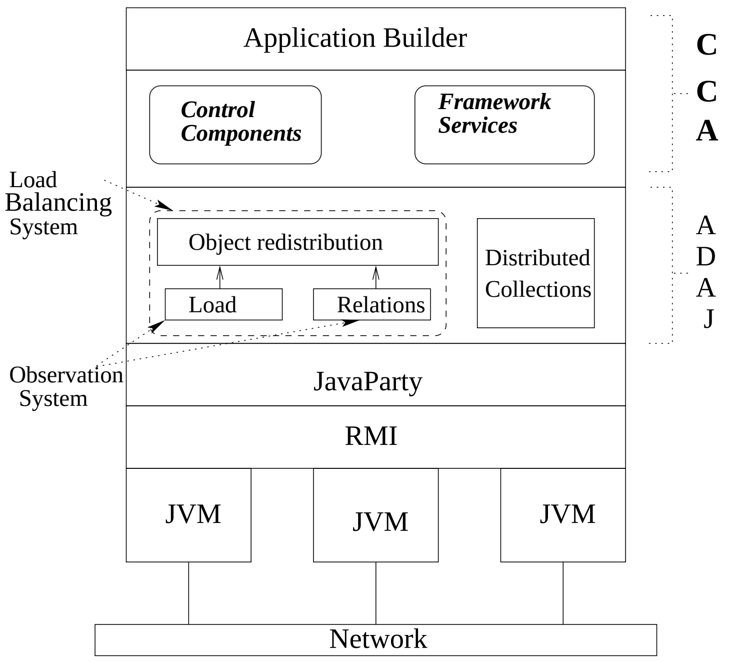

Figure 1: The ADAJ multi-layer structure.

2.2 Observation system in *ADAJ*

The observation system [25, 12], needed for load balancing, consisted of two mechanisms. The first is a mechanism for observation of the dynamic relations between the objects during the execution. The second is a computer load observation mechanism. Note that, knowledge of load of all the computers of the system is insufficient to ensure a correct object distribution. The introduction of a mechanism of observation of the relations (and communications) between selected objects within the JVM makes it possible to enrich knowledge about actual application-related loads during the execution. This observation allows estimation of global object work by counting called method activations. In this context, the *ADAJ* recognizes two object types: *local* and *global* objects.

- **Local objects** are traditional Java objects which belong to the users. They are instantiated in the JVM. Their static part is in the same JVM and they can be used only within the JVM they were instantiated in. If these objects are needed on another JVM, they must be copied. The running state of a local object and its attributes are to be copied out in the new JVM. There is no coherence maintained between the original and the copy. A new static part is also created by copying if necessary. This copy is also a local object.
- **Global objects** are typically ADAJ objects. They can be created remotely in any JVM. They are remotely accessible and migrable.

Observation of relations between objects (through method invocation) allows for building dynamically the graph of the object application. The marking of an object consists of adding a characteristics to an object. One example is the addition of the "migrability" property to the object so that it becomes a migrable object i.e. it can be moved from one JVM to another. To ensure transparency and facility of the object creation, we chose to use marking on the level of the class. The marking of the objects is done implicitly; the marked objects are those which inherit the class *RemoteObject* of JavaParty [29]. This inheritance makes it possible to remotely access appropriately marked objects.

The implementation of the marking process is done by post-compilation techniques. We used a tool of instrumentation of the byte code: JavaClass [15] developed at the Free University of Berlin. JavaClass allows all kinds of transformations of the bytecode: attribute and method additions, method modifications, and per instruction addition. During the post-compilation of an application within the *ADAJ*, the byte codes of all the classes is examined. Classes which inherit the *RemoteObject* are implicitly marked as global classes. The bytecode of these classes is modified by adding suitable marking information.

3 *Agents in Grid* project overview

In the *Agents in Grid* project we perceive the Grid as a distributed environment consisting of users ready to pay for usage of resources and resource owners that offer their resources for sale. Furthermore, we distinguish a *local Grid*, where some form of "organizational control" has been instantiated (e.g. Grid within a company, or Grid service put on sale by Sun Microsystems [6]). Here, there exists a specific entity, which is *responsible* for maintenance of the Grid infrastructure (and in some way assures *Quality of Service*; *QoS*). Furthermore, existence of such entity makes it reasonable to sign *Service Level Agreements* (*SLA*). On the other hand in a *global Grid* there is no entity that has a direct administrative control over resources. These resources belong to anyone; e.g. to individual owners, which makes them relatively low granularity (e.g. single PC's). In this case it is rather difficult to believe that a "business strength" *SLA* can be signed and *QoS* assured. Note that success of projects like SETI@HOME is not a counter-argument here. In this case (and in similar projects) individual results can be obtained in any order and there is no particular time limit on their completion (i.e. there is no interdependency between results). This situation is not acceptable in most cases involving business applications, where jobs have to be executed in a specific order and by a certain deadline. To remedy this situation we proposed a different approach (see, [19, 17, 21, 16, 22, 23, 24] for more details), where:

- Every *resource* as well as every *User* is represented by an agent (*LAgent*)
- Agents work in teams
- Each team has an agent-manager (*LMaster*)
- Each *LMaster* has a mirror-agent (*LMirror*); working together (sharing team-related information) they attempt at assuring long-term persistence of the team
- An agent worker (*LAgent*) joins a team that satisfies specified criteria
- Team accepts agent workers according to its own criteria
- Selecting a team to do the job involves negotiations between *LAgent*s representing *User*s and *LMaster*s representing teams
- The *CIC* (*Client Information Center*) agent plays the role of a central repository where information about all agent teams is stored. Specifically, it contains information about teams that look for workers (who they look for), and teams offering to execute a job (what resources they offer). Utilization of the CIC represents a "yellow page" based approach to matchmaking [32, 18].

Note that, at the current stage, the choice of the machine that will execute the *CIC* agent is out of scope of our considerations. Due to a large number of complicated questions concerning efficiency and scalability of potential solutions, we will have to address this problem in a comprehensive fashion.

However, it can be assumed that in the development phase of the proposed system a solution similar to that discussed in [18] can be utilized.

As a summary, the proposed structure of the *AiG* system has been depicted in Figure 2, in the form of an AML ([14]) Social Diagram. Note that *LAgent* and *Worker* are two roles, where the role of the *LAgent* is the basic one. However, when the *LAgent* joins a team it becomes a *Worker*. Furthermore, the *LAgent* can become (start playing a role of) an *LMirror*, or an *LMaster*.

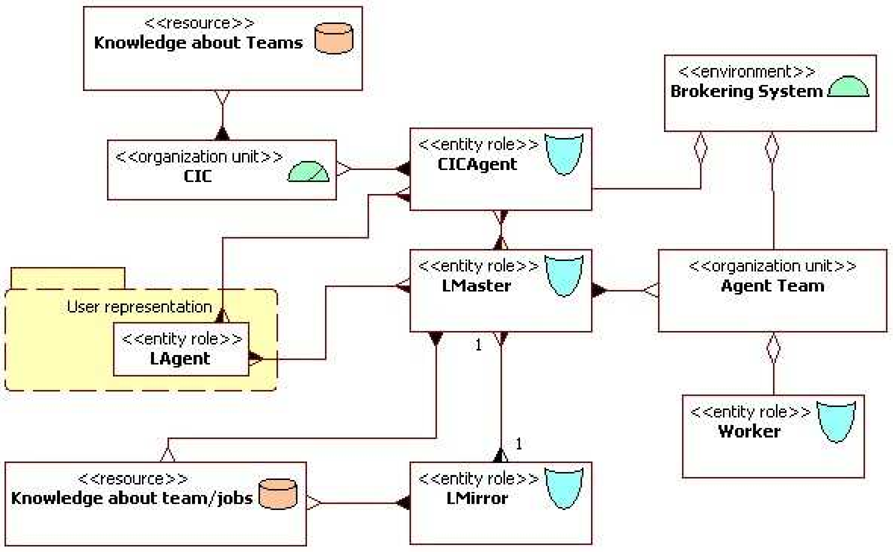

Figure 2: The *AiG* System; AML Social Diagram

4 Integrating *AiG* and *ADAJ* projects

There are two levels of possible integration of software agents and the *ADAJ* project. The first one is integration of such agents directly into the *ADAJ* fabric and utilizing them for functions like load observation and balancing. We will devote to this possibility later parts of this paper. Here let us look into possibility of direct integration of the *ADAJ* and the *AiG* projects.

Observe that both projects represent different "levels of abstraction." The *ADAJ* project is focused on a relatively low-level (object-centered) design, implementation and execution of applications, and (object-level-based) load balancing of individual nodes (computers) that belong to its infrastructure. The *AiG* project, on the other hand, is concerned with providing high-level infrastructure that will allow users to interact with the Grid infrastructure and contract their resources or job execution. Obviously, a resource contracted through the *AiG* infrastructure could be *ADAJ*-based. Similarly, a user-job submitted to the *AiG* could be later executed by the *ADAJ*-running collection of computers. Therefore, it should be easy to envision a situation in which the *AiG* is an agent-based infrastructure, which manages high-level functionalities (the Brain), while the *ADAJ* is (one of possible) Grid-like infrastructure(s) (the Brawn). The text in parenthesis was deliberately put there to show a direct link between the general vision advocated here and content of the seminal paper entitled "Brain Meets Brawn: Why Grid and Agents Need Each Other" ([20]).

What needs to be done to achieve integration at this level is to create an agent-based interface. Such agent, on the one hand, has to be capable of receiving messages from other *AiG* agents (e.g. the *LMaster*). On the on the hand, it has to be able to "communicate" with the *ADAJ* infrastructure, e.g. to dynamically deploy a job, or to pass it in the right form to the right entity to be deployed and executed. On the "way back," it has to be able to receive the results from the *ADAJ* and pass it (wrapped in an ACL message) to the *LMaster*. An initial solution of this type has been outlined in [28].

Note also, that this approach allows the *AiG* to be integrated not only with the *ADAJ* infrastructure. Obviously, an *LAgent* can "represent" a single resource (computer), but it can also be a front-end to other Grid middleware(s). In this way the *AiG* infrastructure may be able to facilitate high-level Grid interoperability, where the responsibility for interacting with independent Grind middlewares will be left to the "gateway agents" representing them (in our case *Worker* agents).

5 Utilizing software agents within the *ADAJ*—overview

Let us now discuss how we can utilize software agents within the *ADAJ* infrastructure. Let us recall that one of key functionalities of the original *ADAJ* system was to facilitate load monitoring and balancing and this is what we decided to utilize software agents for. Proposal presented here is somewhat similar to the one put forward by B. diMartino and collaborators in [10]. In their *MAGDA* system, stationary agents controlled workload, while a mobile agent was responsible for load balancing. One of the important problems with that approach is its heavy reliance on agent mobility, which can be costly. As a matter of fact, one of the reasons for the *AiG* system conceptualized as described in section 3 was to overcome some limitations of the *MAGDA* system. Therefore, we propose a different approach to introduction of software agents (at this stage, primarily, as load managers) into the *ADAJ* system:

- each node (computer) has a *Local Agent* instantiated
- each *Local Agent* utilizes mechanisms developed within the initial *ADAJ* project (see above) to monitor activities within the local JVM
- *Local Agents* in specific time intervals communicate their *local workloads* to the *Central Manager* (specifically, post them to the *blackboard* under its control); this process is similar to bidding in an auction (see also [35])
- the *Central Manager* in specific time intervals calculates *average load* and sends this information as an ACL *INFORM* message to all *Local Agents*
- overloaded *Local Agents* negotiate with underloaded ones transfer of some of their load to them

Let us make a few additional comments. Process described above is completely asynchronous. *Local Agents* post their load information at certain time intervals, however this is a low priority task that should not interfere with their other work. While the *Central Manager* updates the average load at specific time intervals, this process does not depend on having all information updated by all *Local Agents*. Instead, it uses information that is currently available within the blackboard. We also assume that the *Central Manager* may be sending information about change of the average workload only if the change is larger then a certain value (e.g. 5%). This assumption is made to reduce the overall number of messages sent in the system. Finally, note also that it is possible that instead of a push-based informing about average workload (the *Central Manager* sends information to *Local Agents*) it is possible to utilize a pull-based approach in which (*Local Agents* check the average workload when they see a need for this; e.g. when their local workload changes). Finding the most efficient approach for exchanging information about local and average workloads requires further research, including experimental work, similar to that reported in [35].

6 Load balancing

Let us now look into more details of processes involved in utilizing software agents in load balancing within the *ADAJ* infrastructure. Since the *services* that the *ADAJ 2* will be operating on (recall that *ADAJ 2* will be fully *SOA*) consist of objects, we can apply the similar level of granularity of observed processes as in the original *ADAJ* and consider an "object" as a basic entity to be operated on. Taking this into account, in the process of load balancing we can distinguish three basic tasks:

- detecting imbalance
- determining objects to move and the target node for the migration

- migrating selected objects

In the following sections we will specify in some detail how software agents can help in completing these tasks.

6.1 Load monitoring and detecting imbalance

The original *ADAJ* infrastructure incorporated a highly centralized model of observing the load of each machine and detecting the overload or underload of its nodes. Even though the information about the local load of each JVM was generated by a *local Observer module* running on every node, the whole list of objects in the JVM was transferred (through the JavaParty infrastructure) to the *global Observer module* for load analysis. This implies that the *global Observer module* needed to posses updated information not only about local workload, but also about all objects residing on all the machines in the *ADAJ* infrastructure. This solution, though satisfactory for small to medium-sized Grids (especially thanks to the highly efficient implementation of the JavaParty), was likely to cause serious scalability issues for larger systems.

The centralization of knowledge and decision making processes, as well as the amount of communication required for updating the *global Observer module* about every change in the object structure of the local JVM resulted in the solution outlined above. Note that, while it is possible to utilize only an approximate value of average load balancing, to be able to decide which objects are to be moved current information about object localization is absolutely necessary.

In the proposed approach, we will still utilize *local Observer modules* designed and implemented in the original *ADAJ*, but instead of transferring data generated by them to a central location to be analyzed, we will leave decisions to the *Local Agents*. Obviously, being over (or under) loaded depends on the state of all other nodes, while the average load value varies over time. Therefore, we have decided to introduce a *Central Manager*, that collects (within a blackboard) information about load of individual nodes. It should be noted that, even though we are introducing a global entity storing information about all the nodes (the *Central Manager*), this entity differs greatly from the *global Observer module* from the original *ADAJ*—it receives only information about the load level, not about specific objects generating this load. Furthermore, as indicated above, it periodically uses data stored within the blackboard and prepares information about the *estimated* average load. This information is sufficient for all nodes to establish if their workload is high enough to be claimed to be in a state of overload, or low enough to be considered underloaded. Regardless of a specific solution used (push or pull-based approach), after a *Local Agent* obtains/receives the current average load estimate it can compare its own load to this value and make a decision on whether it needs to act (try to off-load some of its work), or if its load can be considered "balanced." Note that a specific value of the difference between the average load and the local load that will initiate an action of the node remains to be established experimentally. Observe also that in the current design of the system only agents that are overloaded will be acting trying to reduce their loads (will be active), while underloaded nodes will be passively awaiting load increasing proposals. It is however conceivable that both sides could be actively seeking ways of balancing their load. We will consider this latter solution in the future.

6.2 Determining which objects to move and where

When the *Local Agent* determines that the node it represents is overloaded and thus decides to move some of its objects to a different one, it needs to find a machine that will accept a specific part of the load. In the original *ADAJ* this functionality was performed by the *global Load module*. Based on the fact that dynamic adaptability of workflows is one of the few areas of successful utilization of software agents in real-life applications (see, [13, 31, 11]), we have decided that this is precisely where software agents can help in increasing scalability and efficiency of the platform. Specifically, in both cases reported in literature software agents were associated with entities placed in a dynamic environment with a goal of managing local workload through utilization of negotiations. In the case of Daimler Chrysler ([13, 31]) agents representing multi-purpose machines in an assembly plant negotiated the flow of parts that were to be produced. In the transport case ([11]), agents representing trucks, load and drivers dynamically negotiated flow of goods. In both cases a performance gain of about 10-20% over static scheduling was reported.

We have, therefore, envisioned a model in which load balancing decisions in the system are made in the course of a multi-agent negotiation (see, also [33, 30]). In this solution the overloaded agent sends a *Call For Proposal* (*CFP*) message to other agents in the system, specifying objects it would like to "give away." Here we plan to use the *FIPA Contract Net Protocol* ([1]) for the negotiation process. However, negotiations may need to be modified to be able to iteratively negotiate with selected partners (which is not included in the original Contract Net, where only a single round of negotiations is expected). Selection of specific objects that the *Local Agent* would like to send to another node should be a result of (multi-criterial) analysis of: (a) object-dependencies, i.e. local and global objects, (b) cost of moving objects (or groups of objects), (c) economical constraints, etc. Agents receiving the *CFP* will utilize (multi-criterial) analysis to establish if accepting these objects "makes sense to them."

Note that we assume here that agents in the system are "benevolent." In other words, we assume that underloaded agents are "very interested" in receiving objects from overloaded agents. Obviously, this assumption reduces (if not completely eliminates) the impact of the economic considerations within the *ADAJ*. However, we would like to claim that this assumption is a reasonable one. Note that the *ADAJ* is a very tightly coupled system and can be considered an example of a "desktop Grid," where all resources belong to a single owner. In this case the assumption about benevolence of agents working together to maximize the throughput is a reasonable one. At the same time, integration of the *ADAJ* with the *AiG* will represent the moment in which the economic model will come to play with full force. Here, the agent representing the *ADAJ*-based infrastructure will try to negotiate the best conditions for selling resources it represents.

When considering the scalability of the proposed approach we need to address the overhead related to sending a *CFP* to all *Local Agents* within the *ADAJ* infrastructure. Let us assume that in the push model information about current average workload has been send to all (n) *Local Agents* in the *ADAJ* infrastructure. Next, k of them have determined that they are overloaded. As a result $k*(n-1)$ *CFP*s are send. Obviously at least $(k-1)*(k-1)$ of these messages are sent completely uselessly (to agents that are overloaded to start with; while, currently, we do not allow for object swapping as a method for performance optimization). To avoid this situation, we propose a solution in which the *Manager Agent*, apart from calculating the estimated average load, also holds a list of underloaded nodes. This can be achieved in a very easy way as each node has a "placeholder" within the blackboard and thus the *Manager Agent* can immediately establish which nodes are underloaded. Therefore, when a *Local Agent* representing an overloaded machine needs to find a receiver for some of its objects, it can query the *Manager Agent* for a narrowed-down list containing only these agents that can be expected to be interested in receiving more work.

Once the agent determines (as a result of negotiations), which objects are to be sent and to which node, the object migration process can commence.

6.3 Object migration

In the original *ADAJ* platform the task of object migration and tracking their location was handled by the JavaParty framework. However, due to reasons described above, we have decided to change this layer of the application. Thus, a need arose to develop or facilitate a solution for for transferring objects and for tracing movement of objects (to update all necessary references once an object moves). We have considered using agents in both of these challenges, but regarding the migration process itself we have found the following disadvantages of such approach:

- To be compliant with FIPA standards ([2]) and taking into account capabilities of existing agent platforms, objects would have to be transfered within ACL messages (only the *Voyager* agent platform introduced in the late 90'th and currently extinct had capabilities for transferring objects directly between agents; [34]), which would limit the scope of possible serialization methods and generate an overhead related to serialization and deserialization.

- The solution would require to be developed from scratch within the JADE ([4]) agent platform (as we are not aware of any off-the-shelf middleware frameworks taking care of such functionality)

We have therefore decided, that the task of moving an object from one machine to another can be more easily implemented using a standard approach, especially given that we have experience in this field gained when developing the original *ADAJ*.

Apart from the migration itself, it is also necessary to be able to trace movement of objects, as local objects depend on such capability of the system. This functionality, on the other hand, necessitates capability of notifying nodes holding dependent objects about migration of their dependencies. Considering this task we have come up with a simple protocol of communication between the nodes engaged in the migration process. This protocol will ensure that the migrating object is not used while it is in a transition state and that the information about its new location is delivered promptly and reliably.

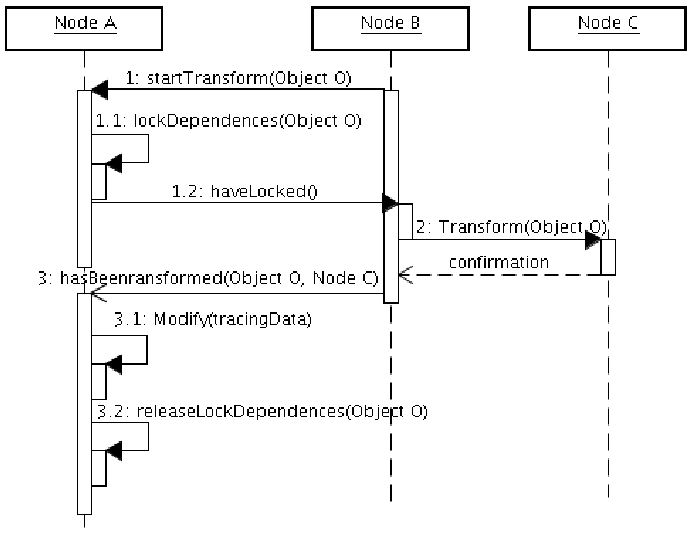
Node A
Node B
Node C
1: startTransform(Object O)
1.1: lockDependences(Object O)
1.2: haveLocked()
2: Transform(Object O)
confirmation
3: hasBeenransformed(Object O, Node C)
3.1: Modify(tracingData)
3.2: releaseLockDependences(Object O)

Figure 3: sequence Diagram

In the diagram we can see 3 nodes (A,B,C):

- Node A holds some objects that depend on the object O that is being transferred
- Node B is overloaded and starts the transfer of object O to node C
- Node C is underloaded and is ready to receive object O

As is shown on the diagram, the protocol specifies the interaction as follows:

- Node B sends a message to Node A telling it about the start of the transfer of object O
- Upon receiving the notification, Node A locks all remote references to object O, to prevent calling O's methods during the transferring process
- When O has been locked, Node A sends a confirmation message to B, to tell it that it is now safe to transfer the object
- Object O is transferred to Node C
- Node C sends a confirmation message to node B, to tell that transformation has been completed
- Node B sends a notification to Node A, telling it that the object reached its destination and providing it with the object's new location
- Node A modifies its tracing data with the new location of object A and releases locks on references of object O's

Again, we have considered the agent-based approach to this problem. The requirement for each node is to have an entity listening to notifications from other nodes and then registering the movement of the dependent objects. We have decided that this is a good place to utilize capabilities of existing software agent platforms (JADE in our case), due to the following reasons:

- JADE agents take the burden of inter-platform communication off the developer—JADE provides a uniform model of sending and receiving messages.
- JADE agents run in a separate thread, hence can be easily used as listeners to messages from the outside of the platform without interfering with the task execution
- JADE agents have built-in queuing of messages, thus enabling better scalability of the solution.

It should be noted, however, that using agents in such a scenario makes it necessary to implement an interface of integration between the non-agent (*ADAJ*) middleware infrastructure, used by the *ADAJ* services for remote method invocation and the listener agents. The possible implementation of such interaction depends mostly on the selection of the agent execution scenario.

- If the agent is to be run separately of the *ADAJ* infrastructure, then the only way for them to interact would be by some means of inter-process communication (such as pipes or shared memory). This would, of course, mean that the infrastructure would need either to constantly monitor the shared memory or check it for notifications on every remote method invocation. This solution, while separating the infrastructure from the inter-host communication method is likely to be burdened with a serious inter-process communication overhead.
- The agent can run in the same process as the *ADAJ* infrastructure. If this is the case, the agent could simply pass the reference to an appropriate module taking care of the object tracking and interact by calling the module's methods directly.

Not determining the final implementation approach (which will be determined experimentally), we can still state that the proposed solution to migration and movement notification lets us clearly separate the infrastructure responsible for object migration, tracking and remote invocation from the agent layer that takes care of providing interested parties with the information about when and to where specific objects are to be transferred.

7 Concluding remarks

In this paper we have considered how software agents can be introduced into the *ADAJ* middleware. Our observations and experiences based on the *Agents in Grids* project allowed us to specify two levels of agent-*ADAJ* integration. The high level that can be used without any direct interference within the *ADAJ* and a low level that infuses software agents into *ADAJ*. We have also outlined how the latter proposal can be actually realized. We are currently investigating which infrastructure can be used to provide flexible and efficient object migration that will also be easily integrable with appropriate parts of the *ADAJ* and JADE agents. We will report on our progress in subsequent publications.

Acknowledgement

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] The foundation of intelligent physical agent (fipa). `http://fipa.org/`.

[3] Remote method invocation home. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

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

[5] Javaparty software. University of Karlsruhe, 2008. `http://svn.ipd.uni-karlsruhe.de/trac/javaparty/`.

[6] Sun utility computing. Sun Microsystems, 2008. `http://www.sun.com/service/sungrid/index.jsp`.

[7] Iyad Alshabani. *A Framework for Distributed and Parallel Software Components*. PhD thesis, University of Lille, Lille, France, December 2006.

[8] Iyad Alshabani, Richard Olejnik, and Bernard Toursel. Parallel tools for a distibuted components framework. In *International Conference On Information and Communication Technologies: From Theory To Applications*, Damascuc, Syria, April 2004.

[9] Iyad Alshabani, Richard Olejnik, Bernard Toursel, Marek Tudruj, and Eryk Laskowski. A framework for desktop grid applications: Ccadaj. In *ISPDC*, pages 208–214, 2006.

[10] Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvatore Venticinque. Magda: A mobile agent based grid architecture. *Journal of Grid Computing*, 4(4):395–412, 2006.

[11] M. Becker, G. Singh, B.-L. Wenning, and C. Gorg. On mobile agents for autonomous logistics: An analysis of mobile agents considering the fan out and sundry strategies. *International Journal of Services Operations and Informatics*, (1), 2007.

[12] Amer Bouchi, Richard Olejnik, and Bernard Toursel. An observation mechanism of distributed objects in Java. In *PDP*, pages 117–122, 2002.

[13] S. Bussmann and K. Schild. An agent-based approach to the control of flexible productionsystems. In *Proceedings 8th IEEE International Conference on Emerging Technologies and Factory Automation*, volume 2, pages 481–488, 2001.

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

[15] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[16] Mateusz Dominiak, Maria Ganzha, Maciej Gawinecki, Wojtek Kuranowski, Marcin Paprzycki, Svetozar Margenov, and Ivan Lirkov. Utilizing agent teams in grid resource brokering. *International Transactions on Systems Science and Applications*, 3(4):296–306, 2008.

[17] Mateusz Dominiak, Maria Ganzha, and Marcin 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, Los Alamitos, CA, 2007. IEEE CS Press.

[18] Mateusz Dominiak, Wojciech Kuranowski, Maciej Gawinecki, Maria Ganzha, and Marcin Paprzycki. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, pages 327–335. PTI Press, 2006.

[19] Mateusz Dominiak, Wojciech Kuranowski, Maciej Gawinecki, Maria Ganzha, and Marcin 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.

[20] Ian Foster, Nicholas R. Jennings, and Carl 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.

[21] Maria Ganzha, Marcin Paprzycki, and Ivan 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.

[22] Wojciech Kuranowski, Maria Ganzha, Maciej Gawinecki, Marcin Paprzycki, Ivan Lirkov, and Svetozar Margenov. Forming and managing agent teams acting as resource brokers in the grid—preliminary considerations. *International Journal of Computational Intelligence Research*, 4(1):9–16, 2008.

[23] Wojciech Kuranowski, Maria Ganzha, Marcin Paprzycki, and Ivan 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.

[24] Wojciech Kuranowski, Marcin Paprzycki, Maria Ganzha, Maciej Gawinecki, Ivan Lirkov, and Svetozar Margenov. Agents as resource brokers in grids—forming agent teams. In *Proceedings of the LSSC Meeting*, LNCS. Springer, 2007.

[25] Richard Olejnik, Amer Bouchi, and Bernard Toursel. A Java object observation policy for load balancing. In *PDPTA*, pages 816–821, 2002.

[26] Richard Olejnik, Amer Bouchi, and Bernard Toursel. An object observation for a Java adaptative distributed application platform. In *PARELEC*, pages 171–176, 2002.

[27] Richard Olejnik, Valerie Fiolet, Iyad Alshabani, and Bernard Toursel. Desktop grid platform for data mining applications. In *International Symposium on Parallel and Distributed Computing, ISPDC-06*, Timisoara, Romania, 2006.

[28] Richard Olejnik, Bernard Toursel, Maria Ganzha, and Marcin Paprzycki. Combining software agents and grid middleware. In C. Cerin and K.-C. Li, editors, *Proceeding of the GPC 2007 Conference*, number 4459 in LNCS, pages 678–685, Berlin, 2007. Springer.

[29] M. Phillippsen and M. Zenger. Javaparty—transparent remote objects in java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vegas, USA, 1997.

[30] C. Preist, N.R. Jennings, and C. Bartolini. A software framework for automated negotiation. In *Proceedings of SELMAS'2004*, pages 213–235. LNCS 3390, Springer Verlag, 2005.

[31] R. Schoop, R. Neubert, and B. Suessmann. Flexible manufacturing control with plc, cnc and software agents. In *Proceedings. 5th International Symposium on Autonomous Decentralized Systems*, pages 365 – 371, 2001.

[32] David Trastour, Claudio Bartolini, and Chris 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.

[33] M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. A plug-in architecture providing dynamic negotiation capabilities for mobile agents. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings MA'98: Mobile Agents*, volume 1477 of *LNCS*, pages 222–236. Springer-Verlag, 1999.

[34] F.-J. Wang, C.-Z. Liao, J.-W Huang, and S.-H. Chen. The seventh ieee workshop on future trends of distributed computing systems. chapter An Agent Platform and Related Issues, page 219. 1999.

[35] Katarzyna Wasilewska, Maciej Gawinecki, Marcin Paprzycki, Maria Ganzha, and Paweł Kobzdej. Optimizing blackboard implementation of agent-conducted auctions. *IADIS International Journal on WWW/INTERNET*, 2008. (to appear).