UWAgents: A Mobile Agent System Optimized for Grid Computing

Munehiro Fukuda Duncan Smith

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011
{*mfukuda, duncans*}*@u.washington.edu*
Phone: 1-425-352-3459 Fax: 1-425-352-5216

Abstract

UWAgents is a Java-based mobile agent system optimized for implementing grid computing infrastructure. Unlike most mobile agent systems, which are used only for job deployment and result collection in grid computing, UWAgents addresses specific requirements for coordinating the entire execution of a parallel and distributed computing application. These includes allocation of process identifiers (or MPI ranks), inter-cluster process allocation, job resumption, and (re-)establishment of inter-process communication.

This paper explains how UWAgents addresses these requirements with its two main features: management of an agent hierarchy and agent migration over multiple clusters. The paper also compares UWAgents with other agent systems in terms of parallel job coordination and job deployment performance.

Keywords: Mobile agents, grid computing, grid middleware

1 Introduction

It has been over a decade since mobile agents first received attention as a potential infrastructure for distributed applications such as electronic commerce, information retrieval, distributed simulation, and network management [13]. Despite their inherent applicability to these domains, mobile agents have struggled to establish a killer application that takes full advantage of code/data mobility and navigational autonomy [10].

In the face of such stagnant conditions, consider grid computing as a potential application for mobile agents. Grid computing applications benefit from the following advantages of mobile agents: their navigational autonomy assists the search for computing resources; their state capturing eases job deployment and migration; their migration reduces communication overhead incurred between client and server machines; and their inherent parallelism makes use of idle computers. Several systems have been proposed to apply mobile agents to grid computing [9, 16, 3]. Their mobile agents, however, take charge of only job deployment to and result collection from remote sites. But this is already possible with conventional programming schemes such as rsh and RPC.

Departing from simple job deployment, mobile or multi agent usage has recently focused on other issues in grid computing such as resource discovery and job scheduling. The former works by dispatching agents to a series of distributed resource databases in order to query and screen for the computing resources best suited to their user applications [11]. The latter implements job scheduling as a form of agent negotiation [2], or has each agent learn the best computing node using job execution trials at different sites [6].

Despite the issues mentioned above, the use of mobile agents is still limited to single or parameter-sweeping jobs, each executed independently. While involved in inter-agent negotiation, each agent takes care only of its own job and does not consider the others. To extensively use mobile agents for grid computing, we focus on coordinating the execution of communicating processes such as an MPI application in teamwork among multiple agents, which includes assigning a different MPI rank to each processor; establishing inter-process communication links even over different clusters; moving a process to another site; and resuming a crashed process, thereby reconnecting broken communication links. However, all of those requirements cannot be fulfilled with existing mobile agents. For this reason, we have developed a new agent execution platform named *UWAgents* and applied it to the infrastructure of our *Agent-Teamwork* grid-computing middleware system [5]. In this paper, we will demonstrate effective job coordination with UWAgents.

The rest of paper is organized as follows: Section 2 is an overview of the UWAgents system; Section 3 discusses its application to grid computing; Section 4 analyzes its effective job coordination from both performance and functional viewpoints; and Section 5 concludes our discussions.

2 Implementation

UWAgents is an execution platform for Java-based mobile agents, each of which is a *uwagent*. This section provides an overview of the UWAgents implementation, focusing on its grid computing-oriented features: hierarchical agent creation, a distributed naming algorithm, cascading termination, and over-cluster agent navigation and communication.

2.1 Programming Model

To run UWAgents, each computing node launches a *UWPlace* daemon that exchanges uwagents with other nodes. Figure 1 shows an example of a uwagent program. Each agent extends the *UWAgent* abstract class (line 1), is injected by a *UWInject* command from the Unix shell, is instantiated as an independent Java thread, starts with *init( )* (line 6), and migrates with *hop( )* to a different site (line 11). The *hop( )* method is based on weak migration that resumes a calling uwagent from the top of a given function (line 14) rather than right after where it was captured (line 12).

Currently, UWAgents has two restrictions: (1) local I/O is not forwarded to migrating agents and thus must be closed before agent migration or termination, and (2) if an agent has instantiated subthreads internally, it is responsible for restarting these threads with *start( )*.

```
1  public class MyAgent extends UWAgent {
2    private MyClass myObject;        // carried together
3    public MyAgent(String args[])   {// created locally
4      myObject = new MyObject( );
5    }
6    public void init( ) {     // executed upon an injection
7      proc( );
8    }
9    public int proc( ) {
10     String args[] = new String[3];
11     hop("n1.uwb.edu", "func", args); // goes to n1
12     return 1;                        // can't reach here.
13   }
14   public void func(String args[]) {  // runs at n1
15         ...;                         // more computation
16   }                                  // end of agent
17 }
```

Figure 1. UWAgents' programming framework

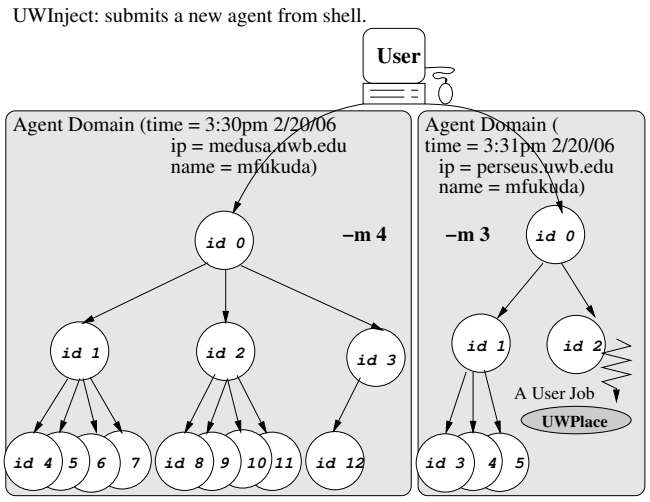

Figure 2. Agent domain

2.2 Agent Management in a Hierarchy

Figure 2 sketches uwagent creation, management, and termination in a hierarchy. Submitted by *UWInject*, a uwagent forms a new agent domain identified by a triplet that consists of an IP address, a timestamp, and a user name. It then becomes the domain root with *id* 0 and can hierarchically spawn descendants using the *spawnChild( )* method. The root agent is also allotted and passes to its descendants the maximum number of children each agent can spawn, (denoted by *m*). By restricting the root agent to creating up to $(m-1)$ children, each agent *i* can identify its children using $id = i * m +$ *seq*, where *seq* is an integer starting from 0 if $i \neq 0$, (i.e, it is not a root) or from 1 if $i = 0$, (i.e., it is a root). As exemplified in Figure 2, agent 1 can create agents $4 \sim 7$ with $m = 4$, while spawning agents $3 \sim 5$ with $m = 3$. This naming scheme requires no global name servers, thus allowing a large number of agents to identify one another easily, for example when coordinating a massively parallel application.

Inter-agent communication is implemented in the *talk( )* and *retrieveNextMessage( )* methods. Together, these methods compose a message as a hash table, transfer it to a destination agent in the same domain by traversing the domain's tree structure, and receive the message at the destination agent using a blocking read. The current UWAgent implementation does not permit inter-domain communication. This is because we do not find strong justification for allowing direct communication between agents that are serving different user applications (i.e., applications submitted separately from the Unix shell) except in two circumstances: resource search and job scheduling. Although some inter-agent negotiation is necessary to acquire the resources and CPU time that agents compete for, this can be handled us-

ing indirect communication to resource databases and CPU schedulers.

UWAgents permits an agent to wait until all of its children terminate or, alternatively, to kill them using the *setTerminationRequest( )* method. The justification for this has to do with how agents communicate with each other. As described above, messages are forwarded along an agent tree where a parent agent and its children share information about their migration. Therefore, a parent must implicitly postpone its termination until all of its descendants have finished their communication and have terminated themselves. This feature helps a parallel application with detecting or enforcing distributed termination.

UWPlace accepts a user job passed from an agent and schedules it as an independent Java thread. Contrary to most mobile agent systems, which launch jobs as Unix processes, UWPlace strictly schedules user jobs using the *suspend( )* and *resume( )* methods provided by Java threads. Scheduling is based on agent runtime attributes such as number of migrations, total execution time, and execution priority.

2.3 Inter-Cluster Migration

UWAgents allows agents to travel between two separate networks or clusters if those networks are linked by one or more gateway machines running UWPlace. Agents on one network can then send messages to those on the other network. In addition to running UWPlace on the gateway machines, the user must specify the IP names of the machines in one of two ways: (1) on the UWPlace command line with the -g switch, in which case only one gateway machine may be specified, or (2) as a parameter to the hop( ) method, which accepts a string array of machine names.

For example, consider agent navigation over a gateway as illustrated in Figure 3. In the command-line option, the following pair of commands indicate that a gateway called *medusa* is available to allow agent *TestAgent* to travel from *hera* to *mnode0*.

```
[user@hera]$ java UWPlace -g medusa &
[user@hera]$ java UWInject hera TestAgent mnode0
```

The same navigation is carried out with the gateway version of the hop method. The following code segment allows an agent to hop from its current location, (i.e., *hera*) to *mnode0*, using one gateway called *medusa*. When it arrives at mnode0, it will call a method called *ArrivalMethod*, with no arguments.

```
String[] gateway = new String[1];
gateway[0] = "medusa";
hop("mnode8", gateway, "ArrivalMethod", null);
```

As shown in Figure 3, a parent agent can deliver a message via *medusa* to its child, even though the child has moved from *mnode0* to *mnode1*. The underlying implementation technique is to have each agent maintain gateway information in its member variables. As an agent hops from the source to the destination node, this gateway information travels along with it. When migrating to a new destination, the agent notifies its parent and children of its new position and gateway information, so that all agents update their own directory of the gateways required to reach their parent and children.

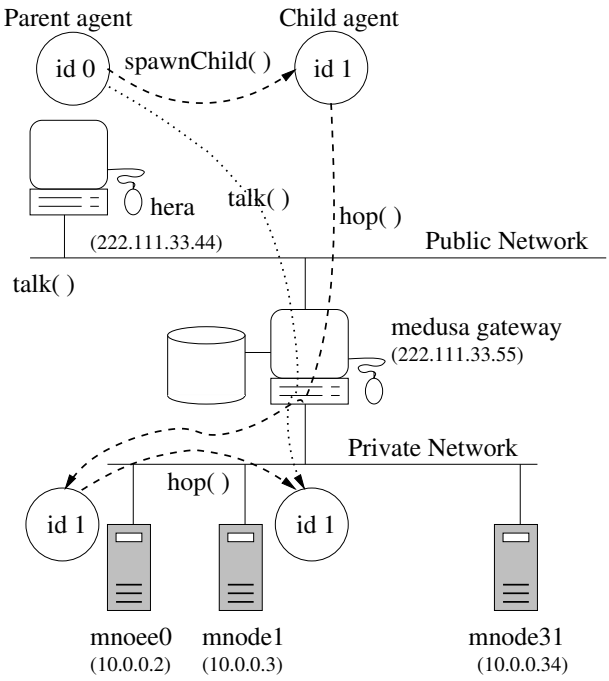

Figure 3. Over-gateway navigation

3 Applications

This section demonstrates how UWAgents is not only applicable to grid computing, but also benefits other applications such as information retrieval, news delivery, and network monitoring.

3.1 Grid Computing

AgentTeamwork is a distributed job-coordinating system for grid-computing applications that we are developing with UWAgents [5].

Figure 4 gives an overview of the AgentTeamwork system. Each member of a group of computer users uploads his/her computing resource information to a shared ftp server, and downloads the AgentTeamwork software, which runs locally. When a user wants to run a new job through AgentTeamwork, it is first preprocessed and enabled for

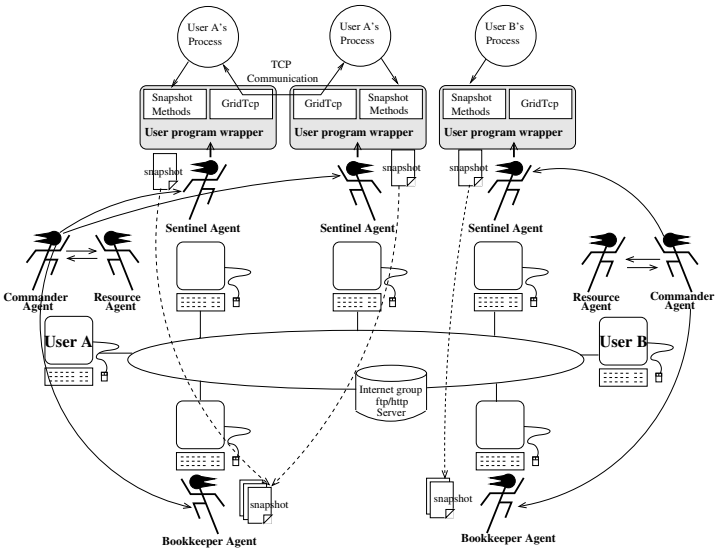

Figure 4. An overview of AgentTeamwork

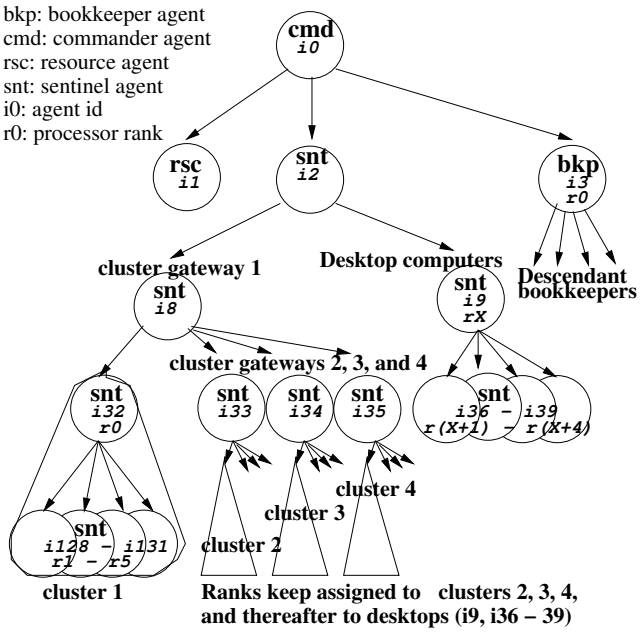

Figure 5. Inter-cluster job distribution in AgentTeamwork

process check-pointing and migration. Several of AgentTeamwork's specialized agents now become involved. A commander agent starts up, and spawns a resource agent. The resource agent checks a local resource database, and calculates a node itinerary based on resource requirements. The commander also hierarchically spawns as many sentinel agents as there are user processes in the job, and as many bookkeeper agents as required by the job.

While the user's job runs, the specialized AgentTeamwork agents work as follows. A sentinel runs the user process at each node, assigns a different MPI rank to the process, monitors it, collects its results, and sends process snapshots to the corresponding bookkeeper, which reroutes them to the other two bookkeepers. The purpose of this snapshot forwarding scheme is to maximize the chance that a current snapshot will be available to resume a crashed agent. Bookkeepers attempt to protect themselves by choosing different destination nodes from their corresponding sentinels, so that both the sentinel and bookkeeper are not killed simultaneously if their node crashes.

Each agent monitors and resumes its parent and children at a different node if they crash. It also migrates to another node in its itinerary if the agent detects that its current location does not satisfy its resource requirements. When the user job generates output, the output is forwarded to the commander, which displays it on the local console.

A modification is currently being made to AgentTeamwork so that it will be able to dispatch a parallel application over multiple clusters using UWAgents' over-gateway navigation. To implement this job distribution, sentinel agents form two subtrees as illustrated in Figure 5: the left subtree is dedicated to clusters, and the right is assigned to single desktop machines. For the left subtree, the leftmost agent at each layer and its descendants are deployed to computing nodes in the same cluster (thus in the same private address domain) while the other agents migrate to a different cluster gateway. This agent-deployment scheme allows sentinels to monitor each other and decide their own MPI rank per cluster. Furthermore, all cluster gateways are managed in the left subtree, which trims off its leftmost subtree at each layer. That makes it easier for each cluster gateway to monitor its own computing nodes in the leftmost subtree as well as the other cluster systems in the other tree branches. Another advantage is that a sentinel stays at each gateway, which allows it to monitor a user application's inter-cluster communication.

3.2 Other Applications

Mobile agents' applicability to and performance in information retrieval were precisely analyzed in [8]. In this application domain, mobile agents can significantly reduce network traffic as well as yield more processing intelligence to servers by transferring the client's programs and executing them at server sites. However, in order to respond to their client, who may move around, mobile agents must know his/her current location. For this reason, the client must obtain a mobile IP; agents must query Yellow Pages servers for the client's IP; or the client program itself must be written to continually inform all its agents of

its current location. UWAgents requires none of these additional services or modification, because it has implemented a location-tracking feature at the system level.

Information delivery is the inverse of information retrieval. For example, consider the hyper news delivery system introduced in [4], in which a news provider dispatches to each subscriber a proxy agent that is responsible for downloading this provider's new articles. Unlike an information retrieval system, this application must frequently deliver information to a large number of dispatched agents. Unless an efficient inter-agent multicast mechanism is used, a provider will repeatedly send the same information to many different proxy agents. The same problem would occur in a data-intensive parallel application that requires an NFS server to deliver files to each process. Since UWAgents forms an agent domain in a tree structure, it can mitigate such delivery overhead by multicasting data along tree branches to all agents.

Network monitoring is another convenient domain for mobile agents [17]. By roaming over a network, mobile agents can detect network events happening at remote routers, gateways, firewalls, and other servers. An obvious problem is that they cannot explore network devices beyond a gateway. This problem is addressed with UWAgents' over-gateway navigation and communication.

In summary, UWAgents benefits network-centric applications and middleware designs with its location-tracking mechanism, tree-based agent domain, and over-cluster agent navigation.

4 Analysis

To differentiate UWAgents from other mobile agent platforms, we have conducted the performance and functional analysis shown below.

4.1 Performance Analysis

For our performance analysis, we used two cluster systems connected to our 100Mbps campus backbone. One is a Myrinet-2000 cluster of eight 2.8GHz Xeons, and the other is a Giga Ethernet cluster of twenty-four 3.2GHz Xeons.

Figure 6 compares migration performance between UWAgents and IBM Aglets [12]. The elapsed time was measured for ten repetitions of agent migration, in which a master agent deploys to a different cluster node 1 to 32 slave agents, each sending back a notification to the master upon its arrival at the destination. While the UWAgents deployment overhead linearly increases as the number of slave agents grows, IBM Aglets has successfully alleviated its overhead increase when deploying 32 agents. This performance gap resulted from a difference in agent-spawning mechanisms: UWAgents has each agent take responsibility

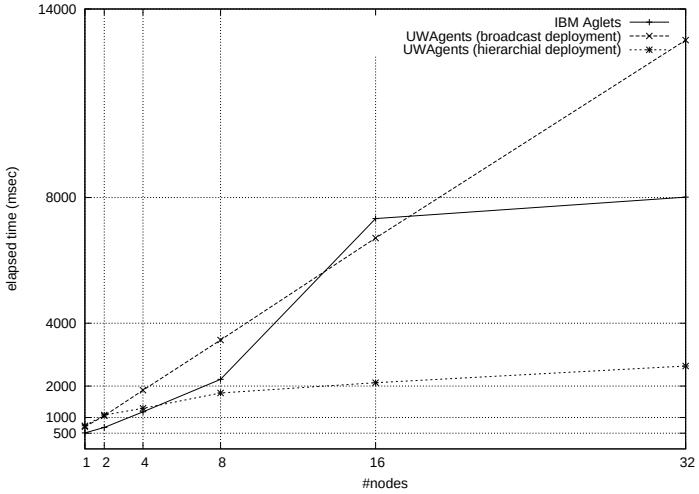

Figure 6. Performance of agent deployment.

for instantiating all of its children in its base class, whereas IBM Aglets performs its child instantiation in a working space named *AgletContext*. This means the server itself deploys new agents concurrently using multi-threading.

For UWAgents, we have also modified our test program to deploy slave agents in a hierarchy. The modification is simple: each agent repeatedly creates a child until its identifier reaches the total number of slaves. Figure 6 shows that, due to the nature of hierarchical deployment, UWAgents slowed down the growth rate of migration overhead logarithmically. Needless to say, UWAgents shows that the more parallelism a job requires, the more important it is to mitigate migration overhead.

4.2 Functional Analysis

Other mobile agent platforms intended for use in a grid computing environment include IBM Aglets [12], Voyager [15], D'Agents [8], and Ara [14]. In the following section, we compare their functionality in terms of agent naming and communication, synchronization, security, and job scheduling.

The first area is agent naming and communication. Consider the AgentTeamwork scenario in which a sentinel agent transfers a snapshot to its corresponding bookkeeper. In order to execute this transfer, the sentinel must first be able to find the correct bookkeeper. D'Agents and Ara give each new agent a server-dependent and unpredictable agent identifier, because of which we cannot systematically correlate the bookkeeper with a given identifier. IBM Aglets uses the AgletFinder agent that registers all agent identifiers.

This requires all agents to report to AgletFinder upon every migration. Voyager identifies an agent through a conventional RPC-naming scheme that must however distinguish all agent names uniquely regardless of their client users. Therefore, each user must carefully choose a unique agent name. The UWAgents naming scheme facilitates this process as follows. Since both the sentinel and the bookkeeper are spawned by the same commander, they are siblings. Their agent identifiers can be calculated by a simple formula based on their position in the agent tree rooted at the commander that is created when the user first injects a job [5]. Sibling agents communicate with each other through their parent, and parent agents automatically delay their termination until all of their siblings have terminated. Combined with the agent ID calculation, this guarantees that the sentinel will be able to find the bookkeeper that it wants to send the snapshot to.

The second area of interest is synchronization. Consider the scenario in which a commander agent wants to wait until all of the sentinel and bookkeeper agents involved in its job have terminated. In IBM Aglets, a parent agent can use the *retract* function to take all of its children back to itself, though the parent still must terminate them one by one. Ara allows all agents to be terminated at once with its *ara_kill* command. Needless to say, the calling agent itself will be terminated as well. In D'Agents and Voyager, we have to implement cascading termination or synchronization at the user level. In UWAgent, because of the way the agent tree is constructed, all of these agents are the descendants of the commander. Therefore, the commander can determine that a job has completed simply by checking the state of its descendants. To implement this check, the commander (or any parent agent) attempts to send a notification message to each of its children, and counts the number of successes. When this number reaches zero, the commander knows that the job has completed. Because of the delayed termination rule described above, each parent only needs to check one level below itself, since it knows that its children will check their children before terminating.

The third area of interest is security. Consider the scenario in which a sentinel agent accidentally or maliciously attempts to communicate with another user's application, such as the bookkeeper agent associated with another sentinel. Although the other mobile agent platforms mentioned above use various security features such as Java bytecode verification, the allowance model, the currency-based model, and the CORBA security service, they have not focused on agent-to-agent communication in a group. SWAT is a mobile agent platform [1] that distributes a cryptographic key to a group of agents using a tree to secure their intra-group communication. This is similar to UWAgents' tree-based distribution of domain information (considered as a cryptographic key). The difference is that SWAT leaves the user to explicitly specify which agents may join a group, whereas UWAgents automatically admits a new agent to its parent's group.

The final area of interest is job scheduling. Since multiple agents are allowed to migrate to the same node, it would be useful to be able to make decisions about when to run processes based on conditions at the current node. As described in Section 2, UWAgents implements a scheduler at each UWPlace. The other mobile agent platforms were released without a native scheduling mechanism, though IBM Aglets now has an add-on called Baglets to address this problem [7].

5 Conclusions

We have focused on distributed job coordination in grid computing as the most promising application domain that uses the code/data mobility and navigational autonomy of mobile agents. Such distributed job coordination requires allocation of sequential process identifiers, (re-)establishment of inter-process communication, inter-cluster process allocation, and job resumption. To fulfill these requirements, we have developed the UWAgents mobile agent execution platform, which forms a hierarchical agent domain (thus easing agent naming, communication, and synchronization), implements over-gateway migration, and schedules jobs at an agent level. From a different perspective, UWAgents could be viewed as a platform to facilitate a self-remapping tree of communicating processes over a processor pool, which is a specialty of mobile agents. Throughout the paper, we have demonstrated UWAgents' preeminence over other agents in such areas of specialization as applicability to grid computing and job-deployment performance.

As mentioned in Section 4.2, UWAgents secures agent-to-agent communication through its agent domain. Using SSL, it can also protect migrating agents and transferred messages from eavesdropping by malicious users. For a further enhancement of UWAgents' security features, we are planning to implement agent-versus-server authentication.

Acknowledgments

This research is being conducted with full support from the National Science Foundation's Middleware Initiative (No.0438193). We are very grateful to Eric Nelson and Koichi Kashiwagi of Ehime University, Japan for their programming contribution to UWAgents at its early stage of development.

References

[1] G. et al. Anderson. A secure wireless agent-based testbed. In *Proc. of the Second IEEE International In-*

formation Assurance Workshop – IWIA'04, pages 19–32. IEEE-CS, April 2004.

[2] Alain Andrieux, Dave Berry, Jon Garibaldi, Stephen Jarvis, Jon MacLaren, Djamila Ouelhadj, and Dave Snelling. Open issues in grid scheduling. Technical paper UKeS-2004-03, National e-Science Centre and the Inter-disciplinary Scheduling Network, Manchester, UK, October 2004.

[3] W. Binder, G. Scrugendo, and J. Hulaas. Towards a secure and efficient model for grid computing using mobile code. In *Proc. of 8th ECOOP Workshop on Mobile Object Systems: Agent Application and New Frontiers*, Malaga, Spain, June 2002.

[4] Ciarán Bryce and Jan Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, Vol.4:359–384, 2001.

[5] Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating grid-computing jobs with mobile agents. *International Journal of Applied Intelligence*, to appear in Special Issue on Agent-Based Grid Computing, 2006.

[6] Aram Galystyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the grid using reinforcement learning. In *Proc. of the 3rd Intenational Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3 (AAMAS'04)*, pages 1314–1315, New York, July 2004. IEEE-CS.

[7] A. Gopalan, S. Saleem, M. Martin, and D. Andresen. Baglets: Adding hierarchical scheduling to aglets. In *Proc. of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 229–235, Los Angeles, CA, August 1999.

[8] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: applications and performance of a mobile-agent system. *Software – Practice and Experience*, Vol.32(No.6):543–573, May 2002.

[9] S. Hariri, M. Djunaedi, Y. Kim, R. P. Nellipudi, A. K. Rajagopalan, P. Vdlamani, and Y. Zhang. CATALINA: A smart application control and management environment. In *Proc. of the 2nd International Workshop on Active Middleware Services – AMS2000*, August 2000.

[10] David Kotz and Bob Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, Vol.33(No.3):7–13, August 1999.

[11] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems. *Software Practice and Experiance*, Vol.32(No.2):135–164, February 2002.

[12] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Professional, 1998.

[13] Milojicic, D. et al. **Mobility** *Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, 1999.

[14] Holger Peine. Application and programming experience with the Ara mobile agent system. *Software – Practice and Experience*, Vol.32(No.6):515–541, May 2002.

[15] Recursion Software Inc. *Voyager ORB Developer's Guide*. Frisco, TX, 2003.

[16] O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *Proc. of the 2nd International Workshop on Active Middleware Services – AMS2000*, August 2000.

[17] Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, Abhijt Pathak, Megan Carey, Murlidhar Koka, and Paul Dokas. Active monitoring of network systems using mobile agents. In *Proc. of Networks 2002, a joing conference of ICWLHN 2002 and ICN 2002*, pages 269–280, Atlanta, GA, August 2002. World Scientific.