AgentTeamwork: Coordinating Grid-Computing Jobs with Mobile Agents

Munehiro Fukuda [1] Koichi Kashiwagi [2] Shinya Kobayashi [2]

[1] Computing & Software Systems, University of Washington, Bothell, *mfukuda@u.washington.edu*

[2] Computer Science, Ehime University, {*kashiwagi, kob*}*@cs.ehime-u.ac.jp*

Abstract

AgentTeamwork is a grid-computing middleware system that dispatches a collection of mobile agents to coordinate a user job over remote computing nodes in a decentralized manner. Its utmost focus is to maintain high availability and dynamic balancing of distributed computing resources to a parallel-computing job. For this purpose, a mobile agent is assigned to each process engaged in the same job, monitors its execution at a different machine, takes its periodical execution snapshot, moves it to a lighter-loaded machine, and resumes it from the latest snapshot upon an accidental crash. The system also restores broken inter-process communication involved in the same job using its error-recoverable socket and mpiJava libraries in collaboration among mobile agents.

We have implemented the first version of our middleware including a mobile agent execution platform, error-recoverable socket and mpiJava API libraries, a job wrapper program, and several types of mobile agents such as commander, resource, sentinel, and bookkeeper agents, each orchestrating, allocating resources to, monitoring and maintaining snapshots of a user process respectively. This paper presents AgentTeamwork's execution model, its implementation techniques, and our performance evaluation using the Java Grande benchmark test programs.

Keywords: grid computing, middleware design, mobile agents, process migration, fault tolerance

1 Introduction

While grid computing has drawn some attention in various applications, it has not yet achieved widespread popularity among all end users. One reason is due to a centralized style of resource and job management, which is widely adopted by many grid-computing middleware systems [8, 14]. Despite its simplicity, centralized middleware has two drawbacks: (1) a powerful central server is essential to manage all slave computing nodes, and (2) applications are inevitably based on the master-slave or so called parameter-sweep programming model. Therefore, such middleware may not sufficiently benefit those who can not access a shared cycle server or who want to develop their applications with various patterns of inter-process communication. One extreme example is common ISP users who may wish to mutually offer their desktop computers for grid computing but are apparently unable to share a public cycle server.

For the last decade, mobile agents have been highlighted as a prospective infrastructure of decentralized systems for information retrieval, e-commerce, and network management [13, 18, 21, 22]. As a matter of fact, several systems have been proposed to use mobile agents as their grid-computing infrastructure [2, 3, 16, 30, 32], claiming the following merits of mobile agents: their navigational autonomy contributes to resource search; their state-capturing eases job migration; their migration reduces communication overhead incurred between their client and server machines; and their inherent parallelism utilizes idle computers. Regardless of these merits, mobile-agent-based systems have not yet outperformed other commercial grid-computing middleware. This is because these mobile-agent-based systems have not completely departed from the master-worker model and thus used their mobile agents simply for job deployment and result collection purposes. Therefore, mobile agents have not been able to provide more than an alternative approach to grid middleware implementation.

In contrast, we apply mobile agents to decentralized coordination of user jobs that do not necessarily fit into the master-worker model and have thus extended to other programming models based on pipelining, heartbeat, and all-reduce types of communication. Our only assumption (and restriction) is that grid-computing users must be still able to share at least one ftp server in order to register their own

computing resources. However, we do not need a central cycle server for job submission and coordination. More specifically, once a user downloads resource information from a common ftp server, s/he can independently deploy mobile agents that are responsible for periodically probing the workload of remote hosts, dispatching a job to the best fitted machines, monitoring and capturing the job execution, migrating the job to a machine with a light load, and resuming the job from the latest snapshot upon its accidental crash[1]. Those agents engaged in the same job form a team to pursue this series of distributed job coordination. In addition to decentralized job management, the use of multi agents also facilitates higher fault tolerance by duplicating and managing job snapshots in a distributed fashion. We have implemented this type of inter-agent collaboration in the AgentTeamwork grid-computing middleware system.

Our technical contribution to grid computing is three-fold: (1) a mobile agent execution platform fitted to grid computing, in particular focusing on agent naming and communication that allow an agent to identify which MPI rank to handle and which agent to send a job snapshot to, (2) a check-pointed, error-recoverable, and location-independent socket-based inter-process communication that establishes connection over different cluster systems and funnels multiple connections to a single IP port, and (3) agent-collaborative algorithms for CPU allocation, execution snapshot maintenance, and job recovery that duplicates a snapshot and resumes a crashed job in a decentralized manner.

This paper covers the AgentTeamwork system in Section 2, discusses our implementation techniques of system components in Section 3, demonstrates AgentTeamwork's competitive performance using the Java Grande benchmark suite in Section 4, differentiates AgentTeamwork from its related work in Section 5, and states our future work and concludes the discussions in Section 6.

2 Execution Model

Figure 1 is a system overview. AgentTeamwork targets an agreed-upon computational community of remote desktop/cluster owners formed with a common ftp (or http) server, (e.g., ftp.tripod.com in our current implementation) that only stores a collection of XML-based user account/resource files and the

[1]The accidental crash in this paper assumes process termination, system shutdown, and power-off by a remote computer owner.

AgentTeamwork software kit. The latter includes the UWAgents mobile agent execution platform, an eXist database manager, all mobile agent programs necessary to coordinate a job execution, and Java packages imported when developing an application. Once each computing node downloads these ftp contents, it runs the UWAgents execution platform in background so as to dispatch and to accept user jobs.

In AgentTeamwork, each user locally submits a job with a commander agent, one of the system-provided mobile agents. This agent starts a resource agent that searches its local XML files for the computing nodes best fitted to the user job's resource requirements. Receiving such candidate nodes, the commander agent spawns as many sentinel and bookkeeper agents as the number of nodes required for the job execution. Each sentinel launches a user program wrapper that starts a user process and records its execution snapshots. Such snapshots are sent to and maintained by the corresponding bookkeeper agent for the purpose of retrieving a user process upon its accidental crash. Each agent can also migrate to a lighter-loaded machine autonomously. At the end, all results are forwarded to the commander agent that thereafter reports them to the client user.

In the following, we give more details of the system configuration, the programming interface, and the job coordination scenario.

2.1 System Configuration

Figure 2 shows the AgentTeamwork execution layers from the top application level to the underlying operating systems. The system facilitates the mpiJava API [20] for high-performance Java applications, while they may call native functions and use socket-based communication as a matter of course. Complying with the original API, AgentTeamwork distinguishes two versions of mpiJava implementation: one is *mpiJava-S* that establishes inter-process communication using the conventional Java socket, and the other is *mpiJava-A* that realizes message-recording and error-recoverable TCP connections using our *GridTcp* socket library. The implementation is user-determined with arguments passed to the *MPJ.Init( )* function. Below mpiJava-S and mpiJava-A is the user program wrapper that periodically serializes a user process into a byte-streamed snapshot and passes it to the local sentinel agent. As described above, the

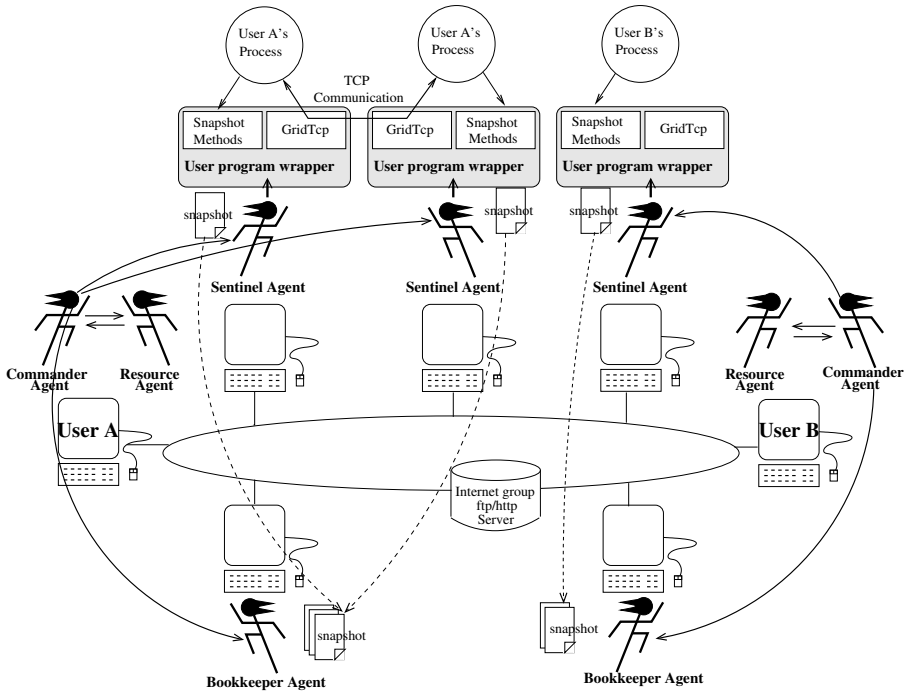


Figure 1. A system overview

sentinel agent sends a new snapshot to its corresponding bookkeeper agent for recovery purposes. All these agents are executed on top of the UWAgents mobile agent execution platform which we have developed with Java as an infrastructure for agent-based grid computing.

2.2 Programming Interface

An application program is coded in the AgentTeamwork-specific framework as shown in Figure 3. If the program uses mpiJava-A, it must include a GridTcp object in a declaration part of system-provided

Java user applications	
mpiJava API	
mpiJava-S	**mpiJava-A**
Java socket	**GridTcp**
User program wrapper	
Commander, resource, sentinel, and bookkeeper agents	
UWAgents mobile agent execution platform	
Operating systems	

Figure 2. AgentTeamwork execution layer

objects (line 4). The code consists of a collection of methods, each of which is named *func_* appended by a 0-based index and which returns the index of the next method to call. The application starts from *func_0*, repeats calling a new method indexed by the return value of its previous method, and ends in the method whose return value is -2, (i.e., *func_2* in this example code). The *MPJ.Init* function invokes mpiJava-A when receiving an ipEntry object that is automatically initialized by the user program wrapper to map an IP name to the corresponding MPI rank (line 8). Following *MPJ.Init*, a user may use any mpiJava functions for inter-process communication (lines 13, 14, 16, and 22). The user program wrapper takes a process snapshot at the end of each function call. Since the GridTcp socket library maintains old MPI messages internally, a process snapshot also contains these messages in it. When the application is moved to or resumed at a new computing node, GridTcp retrieves old messages from the latest snapshot and resends them if they have been lost on their way.

However, framework-based programming restricts the programmability of user applications. To address this problem, we are implementing an ANTLR-based language preprocessor that automatically partitions a given Java application into a collection of *func_* methods.

2.3 Job Coordination Scenario

As illustrated above, a series of job coordination is initiated with a commander agent and carried out in its descendant group of system-provided mobile agents. As shown in Figure 4, the underlying UWAgents execution platform deploys these agents in a hierarchy named an *agent domain*. When injected from the Unix shell prompt, a domain root, (i.e., a commander agent in this case), receives the *id* 0 and the maximum number of children each agent can spawn, denoted as *M* in the following discussions. The root agent can actually create $M - 1$ children through a UWAgents method called *spawnChild( )*, whereas its descendants can further generate *M* children. When a new child is spawned from an agent *i*, it receives $id = i * M + seq$ where *seq* is an integer starting from 0 if $i \neq 0$ or from 1 if $i = 0$. With this naming scheme, the commander agent can distinguish the resource, the first sentinel, and the first bookkeeper agent with *id* 1, 2, and 3 respectively.

Spawned from the commander, a resource agent receives the user's resource requirements, starts an

```
1  public class MyApplication {
2    public GridIpEntry ipEntry[];        // used by the GridTcp socket library
3    public int funcId;                   // used by the user program wrapper
4    public GridTcp tcp;                  // the GridTcp error-recoverable socket
5    public int nprocess;                 // # processors
6    public int myRank;                   // processor id
7    public int func_0(String args[]){    // constructor
8      MPJ.Init( args, ipEntry );         // invoke mpiJava-A
9      .....;                             // more statements to be inserted
10     return 1;                          // calls func_1( )
11   }
12   public int func_1( ) {               // called from func_0
13     if ( MPJ.COMM_WORLD.Rank() == 0 )// if I am rank 0, send data
14       MPJ.COMM_WORLD.Send( ... );
15     else                               // otherwise, receive data
16       MPJ.COMM_WORLD.Recv( ... );
17     .....;                             // more statements to be inserted
18     return 2;                          // calls func_2( )
19   }
20   public int func_2( ) {               // called from func_2, the last function
21     .....;                             // more statements to be inserted
22     MPJ.finalize( );                   // stops mpiJava-A
23     return -2;                         // application terminated
24 } }
```

Figure 3. The AgentTeamwork-specific code framework

eXist database if it has not yet been invoked, searches for XML files best fitted to the requirements, chooses as many available computing nodes as requested by the commander from the top of these XML files, and returns it as an initial itinerary to the commander. It remains alive to probe the workload of remote nodes every five minutes[2] in case the commander further requests another itinerary.

Given a node itinerary, the first sentinel and bookkeeper agents spawn as many descendants as the number of nodes required by the job execution. Each sentinel autonomously calculates an MPI process rank from its agent id using the following formula F1 (see below). It then migrates to a node of the itinerary that is indexed with its rank. Similarly, each bookkeeper obtains its MPI process rank using the formula F2 and migrates to a node indexed with $(N/2 + rank)\ mod\ N$ where *N* is the number of nodes. After launching a user process with its wrapper, a sentinel periodically sends the latest process snapshot to its corresponding bookkeeper agent whose id is calculated with the formula F3. Every time it receives

[2]This probing frequency has been tentatively chosen to prevent the resource agent from overloading network capacity and will be tuned up in our future implementation.

a new snapshot, a bookkeeper agent with $rank = r$ reroutes the snapshot to its two neighbors with their $rank = r - 1$ and $r + 1$, anticipating its accidental crash.

F1: $rank = sId - (2 + (2M - 3)\sum_{i=0}^{\frac{\ln(sId/2)}{\ln(M)}-1} M^i)$

where $rank$ = an MPI process rank, sId = a sentinel id, and M = the maximum number of children each agent can spawn.

F2: $rank = bId - (3 + (3M - 4)\sum_{i=0}^{\frac{\ln(bId/3)}{\ln(M)}-1} M^i)$

where $rank$ = an MPI process rank, bId = a bookkeeper id, and M = the maximum number of children each agent can spawn.

F3: $bId = sId + M^{\frac{\ln(sId)}{\ln(M)}}$

where bId = a bookkeeper id, sId = a sentinel id, M = the maximum number of children each agent can spawn.

These formulas are effectively induced from agent hierarchy. The term $\frac{\ln(id/x)}{\ln(M)} - 1$, where $x = 2$ in F1 and $x = 3$ in F2, indicates which level a given agent of the agent tree resides on, assuming that the first sentinel and bookkeeper are on level 0, and thus $M^{\frac{\ln(id/i)}{\ln(M)}-1}$ defines the number of siblings -1 on the tree level *i*. Hence, $\sum_{i=0}^{\frac{\ln(id/x)}{\ln(M)}-1} M^i$ specifies one less than the total number of agents from the level 0 to the one where this given agent resides. The terms $2M - 3$ in F1 and $3M - 4$ in F2 show how many times the gap in sentinels' and bookkeepers' agent id increase from the last agent on each level to the first agent on the next level. From these terms, we can calculate the MPI rank corresponding to each agent. For the formula F3, $\frac{\ln(sId)}{\ln(M)}$ indicates which tree level a given sentinel resides on, assuming that the first sentinel is on the level 1, and therefore the term $M^{\frac{\ln(sId)}{\ln(M)}}$ specifies the gap from the sentinel to the corresponding bookkeeper in terms of their identifiers. Hence, the bookkeeper id can be obtained by adding the sentinel id, (i.e., *sId* in F3) to this term.

While a user program is being executed, all the system-provided agents monitor the local resource conditions and migrate to another node listed in its itinerary once the current node no longer satisfies its resource requirements. These agents also take charge of monitoring and resuming their parent/child

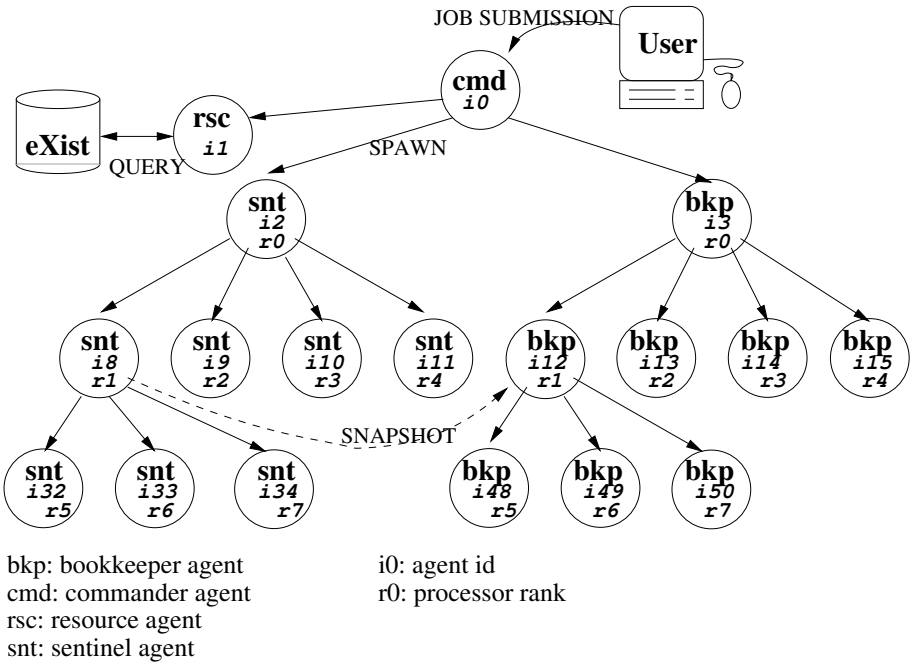


Figure 4. Job coordination in an agent hierarchy

agents from their latest snapshot if they are crashed. If agents exhaust their node itinerary, the commander requests an additional itinerary from the resource agent. Upon a job termination, the standard outputs are all returned as a string array to the commander that then passes the results to the client user through his/her display via email or by files.

3 System Design

This section discusses our implementation techniques for all the system layers enumerated in Figure 2. Our discussion focuses on the three design concerns: a mobile agent execution platform oriented to grid computing (section 3.1), agent-collaborative algorithms for job coordination (section 3.2), and checkpointed computation and communication (section 3.3).

3.1 UWAgents Execution Platform

We have implemented a new Java-based mobile agent execution platform named *UWAgents* that serves as the core infrastructure of the AgentTeamwork system. To run UWAgents, each computing node launches a *UWPlace* daemon that exchanges mobile agents with other nodes. An agent is extended from

the *UWAgent* abstract class, is instantiated as an independent Java thread, starts with *init( )*, and migrates with *hop( )*. The *hop( )* method is based on weak migration and implemented with Java RMI. The *hop( )* method serializes the calling agent into a byte array, sends it with the agent class files to the destination, and resumes the agent from a given user function. 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 sub threads internally, it is responsible to restart these threads with *start( )*.

UWAgents is differentiated from other Java-based mobile agents in the following three capabilities of managing agents and facilitating agent-based grid computing:

1. *Agent naming and communication*: a capability of locating and communicating with a remote agent that keeps migrating to a different node. UWAgents uses a concept of agent domain as explained in Section 2.3. Submitted by the *UWInject* command from a Unix shell prompt, an agent forms a new agent domain identified with a triplet of an IP address, a time stamp, and a user name. It then becomes the domain root with *id* 0 and can spawn descendants. Since each agent *i* identifies its children with $id = i * M +$ *seq* (where *M* is the maximum number of children it can spawn and *seq* is an integer starting from 0 if $i \neq 0$ or from 1 if $i = 0$), UWAgents needs no global name servers. It also enables us to use the formulas defined in Section 2.3. Inter-agent communication is implemented in the *talk( )* method that composes a message of a hash table and transfers it to a destination agent in the same domain by traversing its tree structure. For better performance, the destination IP address is returned and cached in the source agent, which therefore allows the consecutive messages to be sent directly to the destination agent as long as it stays at the same location.

2. *Agent termination*: a capability of waiting for the termination of or instantly terminating all deployed agents, which in turn means a capability of distributed job termination detection. UWAgents permits an agent to wait for the termination of all its children or even to kill them with one 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 keep track of their migration. Therefore, a parent must implicitly postpone its termination until all its descendants have finished their communication and terminated themselves. This feature also helps AgentTeamwork terminate all descendants of a commander agent, (i.e., a root agent) when its client user's application has been finished or aborted. We must however note some amount of overhead incurred by this cascading termination. This is in addition to each agent termination that takes place in the form of interrupting an agent to close all files in use and to wake up all its sub threads in a conditional wait and thereafter killing all these threads. We will examine this overhead in Section 4.3.

3. *Runtime job scheduling*: a capability of scheduling, launching, and pacing a user process execution. UWPlace accepts a user job passed from an agent and schedules it as an independent Java thread. Contrary to most mobile agent systems that launch jobs as Unix processes, UWPlace strictly schedules user jobs with Java thread's *suspend( )* and *resume( )* methods, based on their runtime attributes such as the number of migrations, total execution time, and the execution priority.

3.2 Agent-Collaborative Job Coordination

AgentTeamwork realizes resource allocation, job migration, and job resumption in collaboration among its system-provided mobile agents.

(1) Resource Allocation:

AgentTeamwork uses XML for describing computing resources, an eXist database at each site for managing such XML descriptions locally, and a common ftp server for maintaining them globally. We have chosen XML for the following reasons: (1) each user's computer information must be simply uploaded/downloaded as a text file to/from the common ftp server; (2) eXist is an open-source non-root-manageable database; and (3) the XML DOM parser facilitates our commander and resource agent design that analyzes a submitted job's resource requirements.

Resource allocation in AgentTeamwork is initiated by a commander agent that spawns a resource agent and passes it a set of resource requirements such as the CPU architecture and speed, the memory size, the disk size, the OS type, the network bandwidth, the available time windows, the number of computing nodes *N*, and the multiplier *x* of *N* nodes, where $x = 1$, 1.5, 2, or a larger multiplier. The resource agent launches an eXist database manager if it has not yet started, contacts a common ftp server if the XML files in eXist are outdated, (specifically if more than 24 hours have passed since their last update), passes the commander's resource requirements to eXist as an *XCollection* query, chooses the first *xN* from the top of the best-fitted nodes provided from eXist, and returns this list as a node itinerary to the commander agent.

Each sentinel agent chooses a node from the itinerary indexed with its process rank. On the other hand, each bookkeeper agent migrates to a node indexed with $(N/2 + rank)\ mod\ N$ if the node multiplier *x* is 1 or 1.5. It selects a node indexed with $2N + rank$ if *x* is 2 or larger. With this allocation scheme, a bookkeeper can avoid moving to the same node as its corresponding sentinel. The remaining nodes are saved for future allocation to migrating or resumed agents. If an agent migrates to or is resumed at one of these saved nodes, the agent broadcasts its choice to all the other agents so that they will remove the selected node from their itinerary. Since UWAgents does not guarantee atomic broadcast, two or more agents may collide at the same node, in which case they communicate with UWPlace to find out which has the highest priority. The agents with a lower priority migrate to another node in their itinerary. If agents have exhausted all nodes in their itinerary, the commander agent contacts with the resource agent to obtain another set of *xN* computing nodes.

(2) Job Migration:

A sentinel agent is responsible for moving its user process to a lighter loaded node. As mentioned above, the agent chooses a new node from its itinerary and migrates there with its user process. The migration however breaks up all TCP connections to this process. Other user processes that communicate with this moved process assert an exception to their user program wrapper that makes its sentinel agent wait for a "restart" message to come from the migrating sentinel, restore the wrapper with the new location information, and finally resume the process.

(3) Job Resumption:

A job crash means that the corresponding sentinel agent has also crashed; therefore, we must first resume the crashed sentinel. Two resumption scenarios are considered: one for resuming a child sentinel and the other for a parent sentinel. As shown in Figure 5-A, a child sentinel is resumed through the following five steps: (1) a parent sends a "ping" message to all its children every five seconds[3]; (2) if detecting a child crash, it sends a "search" message to the corresponding bookkeeper agent; (3) the parent receives a "retrieve" message including the crashed sentinel's snapshot from the bookkeeper; (4) the parent sends this snapshot to a new node's UWAgents platform that resumes the crashed sentinel; and (5) the resumed sentinel sends a "restart" message to all the other sentinels that have waited for broken connections to be restored.

Figure 5-B shows a parent sentinel resumed in five steps; (1) a child sentinel receives a "ping" message from its parent every five seconds; (2) if there are no pings delivered for $5 \times agent\_id$ seconds, it concludes that all its ancestors are dead and thus sends a "search" message to the corresponding bookkeeper agent; and (3) through to (5) are the same as those in the child resumption scenario. The parent resumption needs a long period of time to detect a parent crash in the step 2. If all the ancestors of a given child are crashed, this overhead must be unnecessarily repeated until the root agent is resumed. As an improvement, we allow a resumed parent to assume that its ancestors are all crashed, thus to skip the step 2, and to resume its parent immediately.

The resumption scenarios for bookkeeper and commander agents are much simpler. A bookkeeper with rank r is resumed from the one with rank $r + 1$ or $r - 1$. The commander agent is simply restarted by any of its child agent. These two scenarios do not need resumption step 5.

3.3 Check-Pointed Computation and Communication

Snapshots of process states and in-transit messages are handled with AgentTeamwork's user program wrapper and GridTcp socket. On top of them, we have implemented the MPJ package that provides

[3]This interval is tentatively defined to prevent CPU and network resources from being wasted by an agent whose maximum number of children, *M* is large.

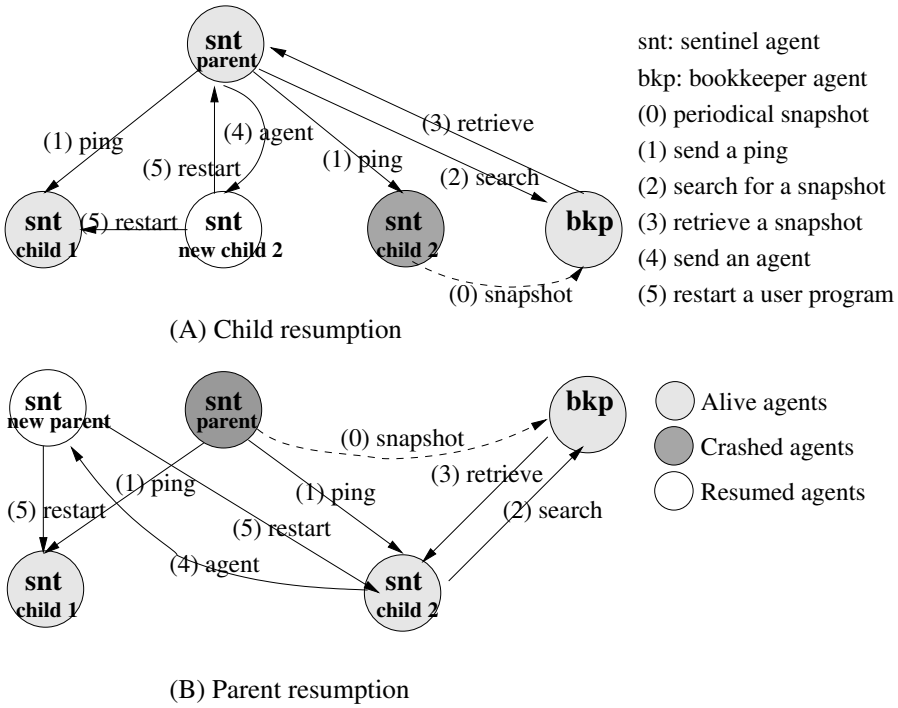


Figure 5. Agent resumption

users with a fault-tolerant MPI environment in Java.

(1) User Program Wrapper:

Using the Java object serialization, the user program wrapper periodically converts all user objects into a byte-presented stream as an execution snapshot. The problem is that Java does not serialize an application's program counter and a stack. To handle this problem, we partition a user program into a collection of methods, each named *func_n* (where n is an integer starting from 0) and returning the index of the next method to call (as shown in Figure 3). In this scheme, the user program wrapper schedules the invocation of these functions and takes a snapshot at the end of each function call.

This solution, however, burdens users with framework-based programming. We will address this burden by extending the UCI Messengers' compiler technique that converts a user program into a series of functions [33]. Figure 6 shows the simplest check-pointing example where nine sequential statements are partitioned into three functions, named *func_0*, *func_1*, and *func_2*, each returning the index of the next function to call. The user program wrapper saves this index value as well as all the user data members, and thereafter calls the indexed function. Upon a process crash, the wrapper retrieves the last

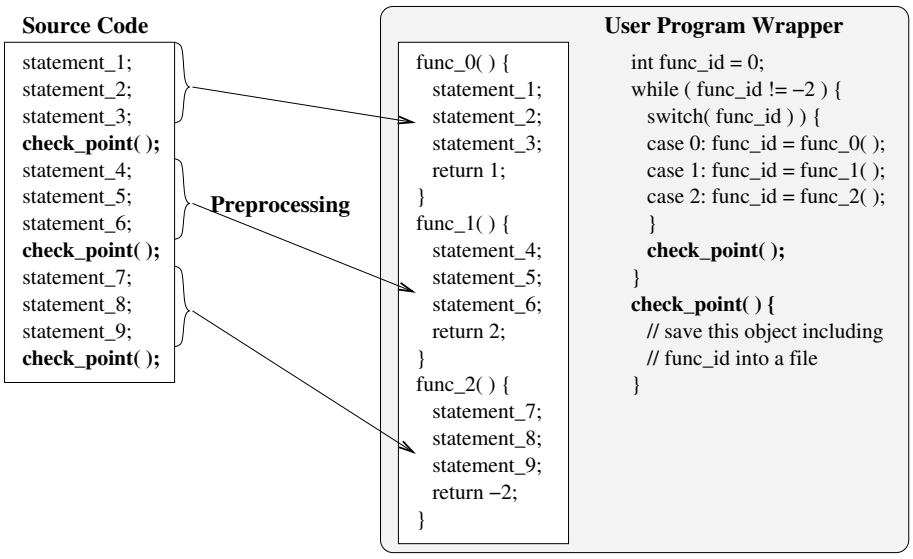


Figure 6. Source program preprocessing

index from the corresponding snapshot and resumes the process from the indexed function.

The following explains our preprocessing strategies for more complicated code. If an *if-then* clause must be partitioned in its middle, we generate three functions: the first function starts from the *if* conditional branch and continues to the break point; the second function resumes the rest of the statements after the break point in the *if* clause; and the third function executes the rest of *else* clause. In order to partition a *while* loop that includes an *if* clause, we prepare another function that includes the same loop but starts from the break point in the *if* clause. The preprocessor can also partition nested functions after applying in-lined expansion to them. The further details of our preprocessing techniques are described in [26, 33].

We have to note that three drawbacks remain: (1) no recursion is allowed due to in-lined function expansion; (2) source line numbers indicated upon compile and runtime errors are not useful since the original source code is completely preprocessed; and (3) a user may still need to indicate snapshot points in his/her application to minimize snapshot overheads.

(2) GridTcp Class:

GridTcp provides message-recording and error-recoverable TCP connections. It saves each connection's incoming and outgoing messages in its internal queues. The queues are captured as a part of a process

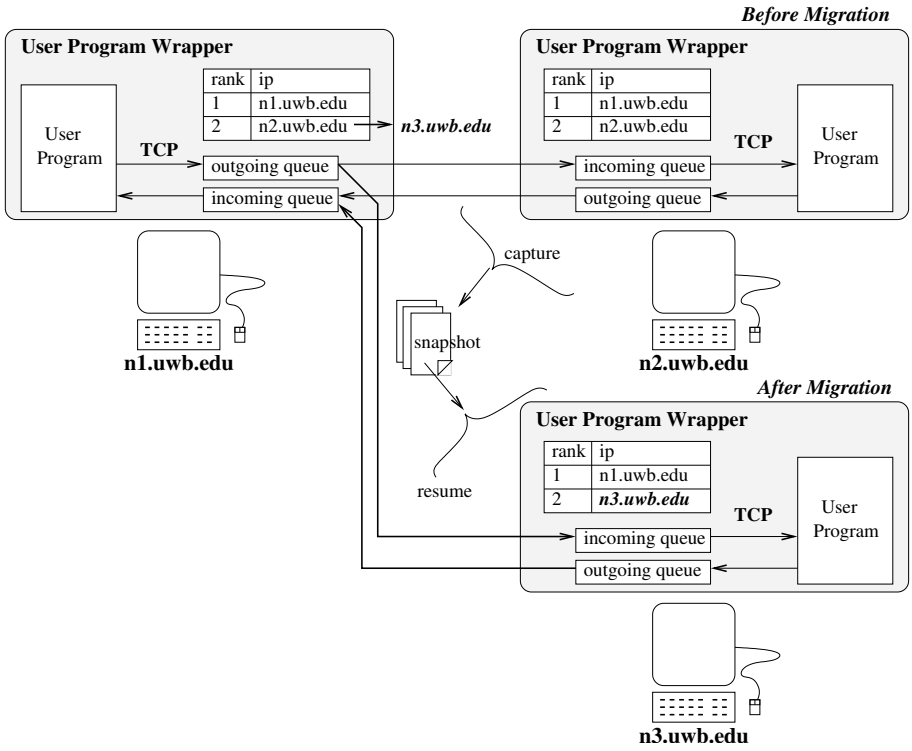


Figure 7. TCP maintenance upon migration

snapshot, and thus retrieved as part of every process migration or resumption. GridTcp garbage-collects its history of these in-transit messages by exchanging a *commitment* message with its neighboring nodes. Our current implementation sends a commitment whenever the underlying user program wrapper takes a new snapshot. Similar to *ssh*, GridTcp also has a capability of not only funneling all TCP connections to a single IP port but also extending connections from one cluster to another. To establish such inter-cluster connections, GridTcp must be executed at each cluster gateway.

Figure 7 illustrates an example of two processes engaged in the same application, running at n1 and n2 of the *uwb.edu* domain, and identified with rank 1 and 2 respectively. When one of the processes migrates from n2 to n3, their TCP connection is broken, which thus asserts an exception to the underlying layers, (i.e., the user program wrapper and the sentinel agent). The sentinel agent with rank 1 receives a new IP address from the one with rank 2, (i.e., n3). Thereafter, these two sentinels pass the *rank2/n3* pair to their user program wrapper that has GridTcp restore in-transit messages from the latest snapshot, update its routing table, and reestablish the previous TCP connection.

(3) MPJ Package:

This is a collection of Java programs that interface MPI applications with the underlying Java socket or GridTcp socket communication. It includes:

1. *mpjrun.java*: allows an mpiJava application to start a standalone execution (i.e., without using AgentTeamwork) by launching an ssh process at each of remote nodes that are listed in the "hosts" file.
2. *MPJ.java*: establishes a complete network of the underlying socket connections among all processes engaged in the same job.
3. *JavaComm.java and GridComm.java*: create and maintain a table of Java or GridTcp sockets, each corresponding to a different processor rank.
4. *Communicator.java*: implements all message-passing functions such as *Send( ), Recv( ), Barrier( ), Bcast( ), Pack( )* and *Unpack( )*.
5. *MPJMessage.java and Status.java*: return the status of the last exchanged message.

The heart of the MPJ package is MPJ.java and Communicator.java. The former, upon an *MPJ.Init( )* invocation, chooses Java or GridTcp sockets, establishes a socket connection from all slave nodes (with rank 1 or higher) to the master (with rank 0), exchanges rank information among all the nodes, and finally makes a connection from each slave to all the lower-ranked slaves. The latter takes care of serializing objects into byte-streamed outgoing messages, exchanging them through its underlying Java or GridTcp sockets, and de-serializing incoming messages to appropriate objects.

4 Performance Evaluation

The following summarizes our latest implementation status. So far, we have:

- Completed all the proposed features of the UWAgents execution platform;
- Implemented agent-collaborative job coordination with the commander, the resource, the sentinel, and the bookkeeper agents;

- Completed GridTcp and the user program wrapper;
- Not yet completed an ANTLR-based language preprocessor, because of which applications must be manually partitioned into a collection of *func_n* methods that are executed and check-pointed with the user program wrapper;
- Implemented seven major functions of mpiJava including *MPI.Send( ), Recv( ), SendRecv( ), Barrier, Bcast( ), Pack( )* and *UnPack( )*; and
- Ported the Java Grande MPJ benchmark programs [28] to AgentTeamwork.

Although AgentTeamwork remains as an on-going project, we have evaluated the degree of parallelism obtained from AgentTeamwork by measuring its performance with our own test programs as well as with the Java Grande MPJ benchmark set. Our performance evaluation has used two cluster systems: a Myrinet-2000 cluster of eight DELL workstations and a Giga Ethernet cluster of 24 DELL computing nodes. Each workstation of the Myrinet cluster has a 2.8GHz Pentium-4 Xeon processor, 512MB memory, and a 60GB SCSI hard disk. Each computing node of the Giga Ethernet cluster has a 3.2GHz Xeon processor, 512MB memory and a 36GB SCSI hard disk. In this section, we will compare the performance between AgentTeamwork and the corresponding executions that use the original mpiJava library [20] and/or the conventional Java socket.

4.1 Inter-Process Communication

We have evaluated how much overhead is incurred by additional communication layers included in AgentTeamwork such as the GridTcp library and the mpiJava-A API. Our evaluation used the PingPong program from the Java Grande MPJ benchmark that measures point-to-point communication performance by sending a message back and forth between two different computing nodes as the message size increases from 1K to 1,024K bytes. Six different executions were measured: (1) mpiJava: executed with the original mpiJava library, (2) Java: converted to use Java sockets and executed with JVM, (3) GridTcp: converted to use GridTcp sockets, (4) GridTcp-1G: GridTcp with another cluster node inserted as a gateway that relays all messages exchanged between two end nodes, (5) GridTcp-2G: GridTcp with

two cluster nodes inserted as gateways where all messages must be relayed twice before delivery to the final destination, and (6) mpiJava-A: executed on top of mpiJava-A.

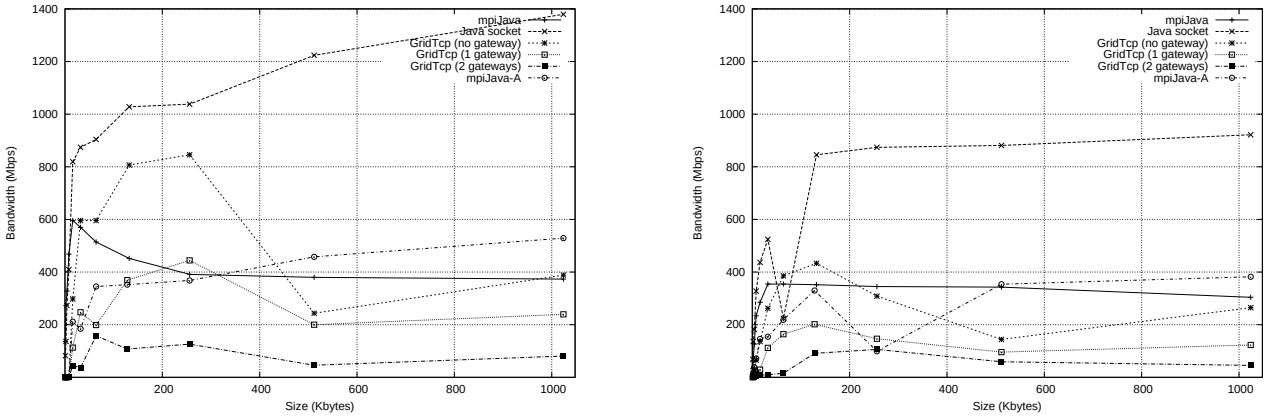
Figures 8-A and 8-B compare the performance of these six versions executed on the Myrinet and Giga Ethernet clusters respectively. Java demonstrated its ideal network bandwidth on both clusters. On other hand, GridTcp increased its bandwidth up through 128K-byte and 256K-byte message transfers on Giga Ethernet and Myrinet respectively, but decreased its bandwidth to 30% of its peak performance on both clusters. Such performance degradation is inevitable, since GridTcp emulates Java sockets and funnels all TCP connections into one connection. GridTcp uses one additional Java thread that reads incoming TCP messages into its queue whenever a user program's main thread is blocked to write large messages. Frequent context switches between the user and GridTcp threads are considered to incur such communication overhead especially for large message transfers.

The original mpiJava showed its 350Mbps and 400Mbps constant bandwidth on Giga Ethernet and Myrinet, whereas mpiJava-A gradually increased its bandwidth and reached 382Mbps and 528Mbps at 1024K-byte message transfers on these clusters respectively, which was actually faster than the original mpiJava. This is because the original mpiJava allocates a new buffer for a message and passes it to the underlying MPI-CH library, which incurs buffer-copying and native-code invocation overheads. Although mpiJava-A inherits all GridTcp overheads, it also fragments a larger message in static buffers, paces their transfers, and thus prevents an excessive number of context switches between the user and GridTcp threads.

GridTcp-1G and GridTcp-2G assume TCP connections established between two different clusters. The figures show that gateway insertions gave a substantial performance drawback as anticipated, which was convincing enough to avoid mapping communication-burst applications over different clusters.

4.2 Computational Granularity

Our next performance evaluation has measured computational granularity when executing three different test programs with mpiJava-A: *MasterSlave*, *Hearbeat*, and *Broadcast*. They repeat a set of floating-point computations followed by inter-processor communication and an execution snapshot. The



(A) Myrinet

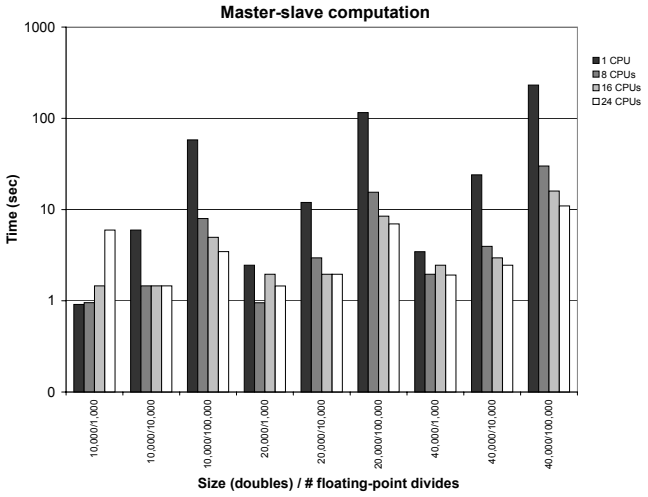
(B) Giga Ethernet

Figure 8. Communication performance

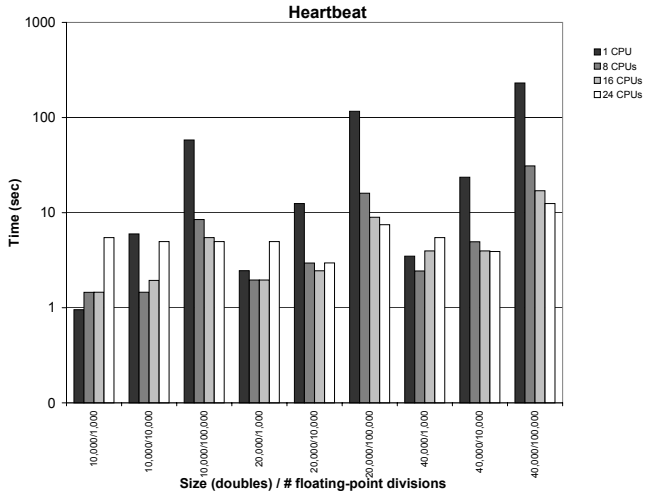
computation is a cyclic division onto each element of a given double-type array. For instance, if they repeat 1,000 divisions onto 10,000 doubles using P computing nodes, their computational granularity is $10,000,000/P$ divisions per set. Each of these programs has a different communication pattern: (1) in MasterSlave, the master and each slave node exchange their local data; (2) in Heartbeat, each node exchanges their local data with its left and right neighbors; and (3) in Broadcast, each node broadcasts the entire data set to all the other nodes. MasterSlave involves the lightest communication overhead, while Broadcast incurs the heaviest communication overhead.

Figures 9 shows the computational granularity and data size necessary to parallelize MasterSlave, Heartbeat, and Broadcast. In order to use 24 CPUs effectively, MasterSlave and Heartbeat need 100,000 floating-point divisions or 40,000 doubles per each communication and snapshot. On the other hand, Broadcast is too communication intensive to scale up to 24 nodes. With 100,000 floating-point divisions, Broadcast's upper-bound is 16 CPUs.

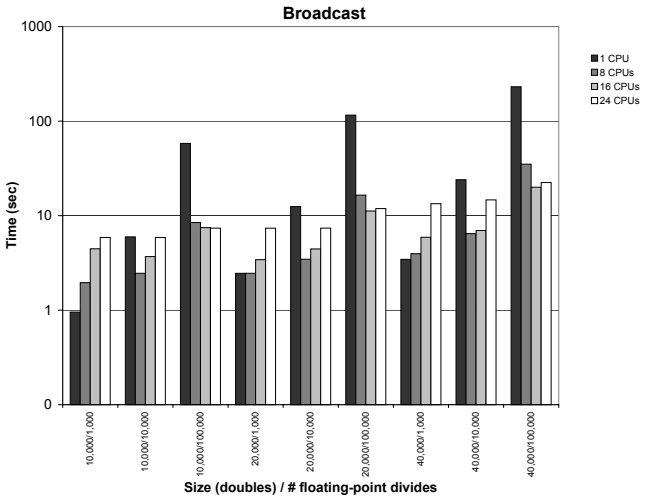
In summary, we expect that AgentTeamwork can demonstrate its scalable performance not only for conventional master-slave applications but also those based on heartbeat (and pipelining) communication such as distributed simulation.



(A) Master slave



(B) Heartbeat



(C) Broadcast

Figure 9. Computational granularity

Program	Section	Description	Data Size
Series	2	Fourier coefficient analysis	40,032 doubles
RayTracer	3	3D ray tracer	600×600 pixels (360,000 doubles)
MolDyn	3	Molecular dynamics simulation	8,788 particles ($8,788 \times 9 = 79,092$ doubles)

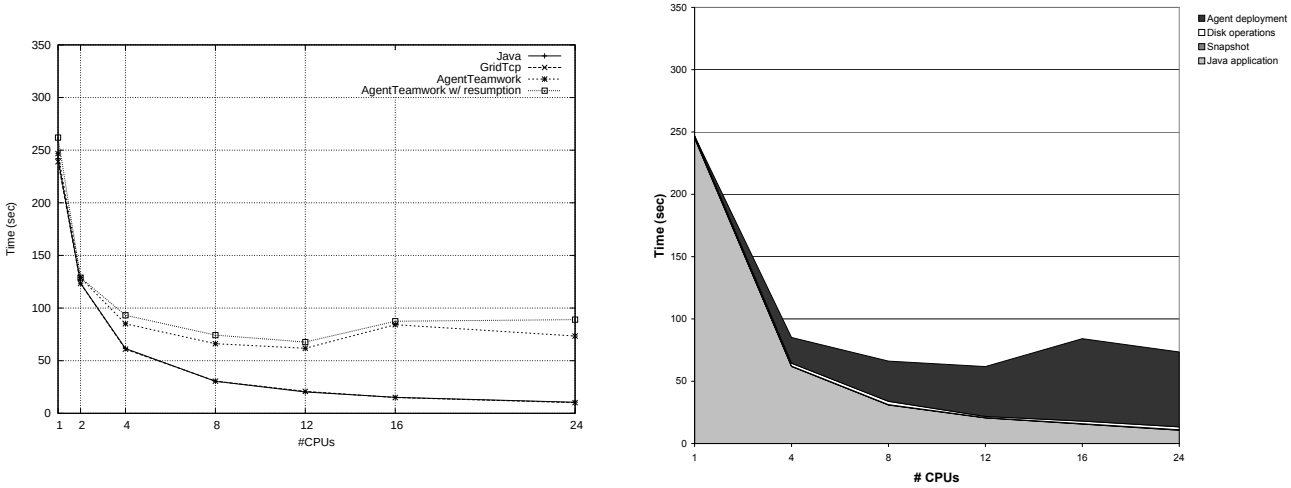
Table 1. Sections 2 and 3 of Java Grande MPJ benchmark suite

4.3 Benchmark Programs

Using the Java Grande MPJ benchmark suite, we have compared AgentTeamwork with other executions and analyzed AgentTeamwork's overhead.

Since mpiJava-A has not yet implemented all MPI functions, we have ported those programs to AgentTeamwork by applying these conversions: (1) Java version: the original programs were converted to use conventional Java sockets and executed over a set of statically chosen computing nodes, (2) GridTcp version: the Java version was further modified to use GridTcp sockets and executed over a set of static nodes, and (3) AgentTeamwork: the GridTcp version was dispatched with AgentTeamwork's mobile agents and executed over a set of computing nodes dynamically chosen. In addition, each converted program runs three times and takes an execution snapshot in both the GridTcp and AgentTeamwork versions.

Among eight benchmarks in Java Grande MPJ's sections 2 and 3, we have focused on the three programs (Series, RayTracer, and MolDyn) summarized in Table 1. Their data sizes are adjusted so that their single-processor execution completes around the range of 250 and 300 seconds. This corresponds to the computational granularity of 100,000 floating-point divisions performed on 40,000 doubles discussed in Section 4.2. We made our benchmark selection based on the following two criteria. First, all benchmarks except Series and RayTracer involve frequent *broadcast/allreduce*-type communication or integer/long-based computation and therefore it is not worth while to increase their data size. In fact, our Java and GridTcp versions did not demonstrate their scalability even with eight CPUs. The other reason is that, despite its *allreduce* communication, MolDyn is a distributed simulation program and therefore one of our target applications. Analysis of MolDyn will contribute to AgentTeamwork performance improvement.

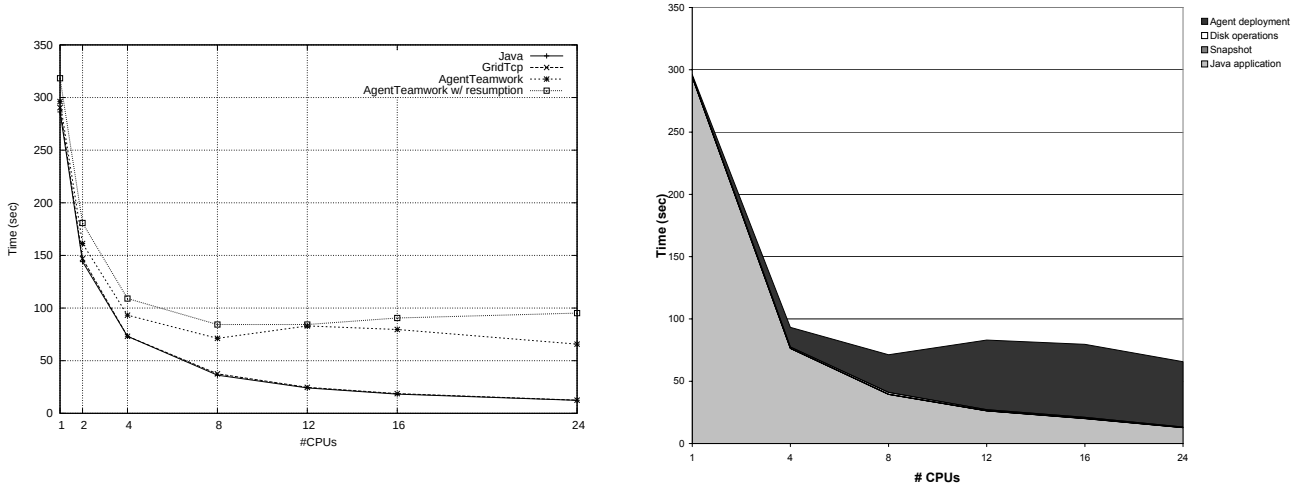


(A) Execution time (B) Performance analysis

Figure 10. Series: Fourier coefficient analysis

Figure 10-A compares the execution time of Java, GridTcp and AgentTeamwork when running Series that computes $40,032/$ $\#nodes$ coefficients at a different processor and collects all results at the master node. Since this is a master-slave program, both Java and GridTcp have demonstrated ideal performance, while performance improvement gradually slowed for AgentTeamwork as the number of computing nodes increased. Figure 10-B depicts an analysis of AgentTeamwork overhead. Although agent deployment dominates the entire overhead, its overhead is stable around 60 seconds in a transition from 16 to 24 nodes. This is the positive effect of deploying agents in a hierarchy.

Figure 11-A depicts an execution of RayTracer that renders a 64-sphere scene at 600×600 pixels with multiple processors, calls *MPJ.allreduce( )* to compute a global checksum, and collects results at rank 0. As with Series, Java and GridTcp's performance results were almost identical, whereas AgentTeamwork increased its substantial overhead. However, RayTracer's snapshot and communication overheads are actually smaller than Series. This occurs because the allreduce operation is only applied to the checksum variable, collected results are integer-type screen variables, and pixels are locally generated. Due to these performance merits, snapshot/disk-writing overheads are almost invisible for AgentTeamwork, as shown in Figure 11-B. The results also demonstrated that the agent deployment overhead remained around 55



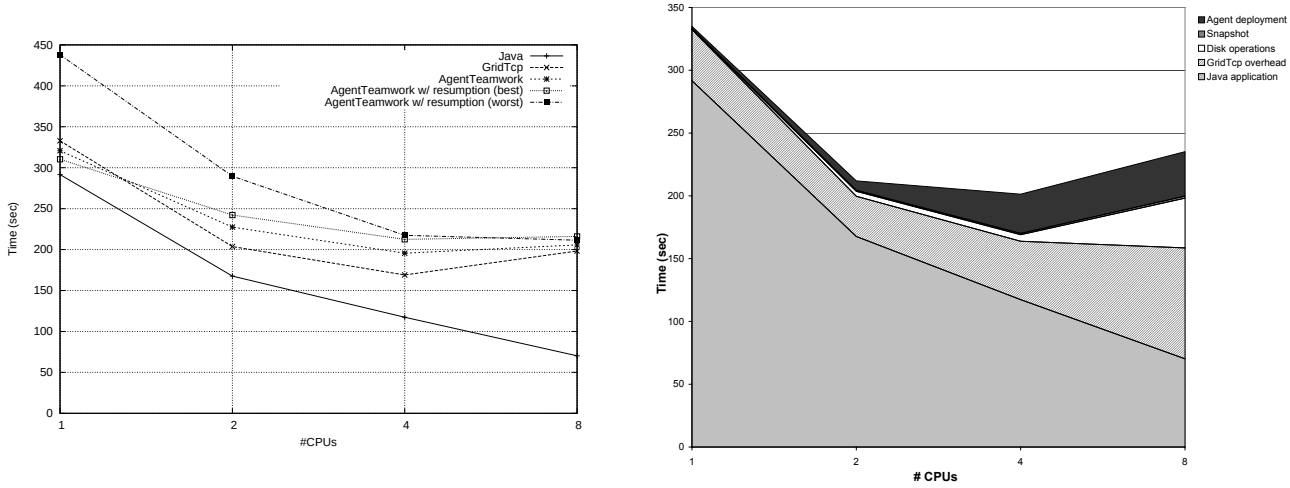
(A) Execution time

(B) Performance analysis

Figure 11. RayTracer: 3D ray tracer

seconds beyond a 12-node execution.

Figure 12-A presents execution results of MolDyn that models 8,788 particles interacting under the Lennard-Jones potential in a cubic space, calculates a particle force for 50 cycles, and exchanges the entire space information among processors every cycle. Unlike the previous two benchmarks, GridTcp and AgentTeamwork displayed increasing communication and agent-deployment overhead respectively, whereas Java demonstrated scalable performance. As shown in Figure 12-B, we have also measured each of AgentTeamwork's overhead factors. (Note that some overheads actually overlapped since the sentinel agent performs check-pointing, disk-writing, and application execution with multiple threads.) Although the agent deployment overhead was again upper bounded at approximately 50 seconds, the GridTcp communication and disk operations have increased their overhead. The main reason is that MolDyn resembles the Broadcast test program in Section 4.2, (i.e., our worst-case test program). Although a sub-simulation space allocated to each node is considerably larger, the original benchmark unnecessarily forces each node to broadcast the entire collection of spatial data to all the other nodes. We expect that MolDyn will demonstrate its scalable performance on AgentTeamwork, once it is rewritten to direct each computing node to send only its local data to the others or just its left/right neighbors.



(A) Execution time (B) Performance analysis

Figure 12. MolDyn: Molecular dynamics

As our last experiment, we intentionally crashed a sentinel agent while we were running the Agent-Teamwork version of Series, RayTracer and MolDyn. We expected AgentTeamwork to automatically resume the crashed agent at a new node and our expectation was met. The experiment killed the sentinel with rank 0 immediately after the agent had taken an execution snapshot. This was done so we could focus on job resumption overheads, although rank 0, as the master process, affects TCP connections with all the other processes. Figure 13 shows the additional time required for a process resumption (mostly between 5 and 20 seconds) when we used 1 to 16 computing nodes. However, the overhead increased with 24 nodes since rank 0 had to re-establish TCP connections with all the other processes. Note that MolDyn's sequential program completed faster with a sentinel crash than a normal execution. This is because the resumed computation no longer needed to share computing resources with a bookkeeper agent.

The worst-case scenario is an agent crash just prior to a checkpoint. We anticipated that such an overhead would be those shown in Figure 13 plus execution time between two consecutive snapshots. However, as plotted in Figure 12-A as MolDyn's worst-case resumption, the actual overhead was better than our estimation when using two or more processors. Since GridTcp has saved previous inter-process

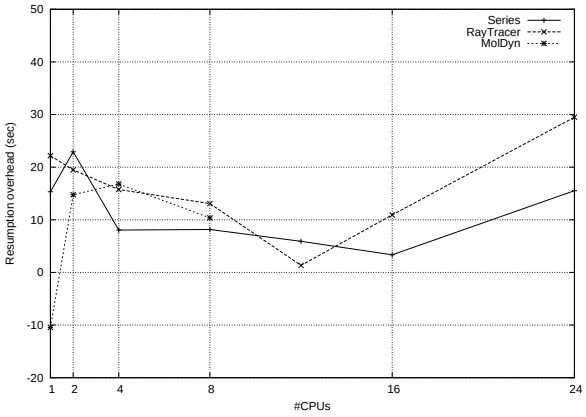


Figure 13. Overhead of process resumption

messages in its snapshot, the restarted process can recover the lost computation without contacting the other processes.

In summary, three requirements must be met to make AgentTeamwork competitive: (1) the computational granularity needs 40,000 doubles $\times$ 10,000 floating-points in order to hide communication and snapshot overheads; (2) applications should avoid combinations of massive and collective communication as illustrated in our Broadcast test program; and (3) the execution length needs to be at least three times larger than the computational granularity to compensate for agent deployment overhead.

5 Related Work

In this section, we differentiate AgentTeamwork from its related work in terms of (1) system architecture, (2) mobile agent execution platforms, (3) job scheduling, (4) snapshot maintenance, and (5) check pointing.

5.1 System Architecture

Globus[9] is the most well known toolkit used to implement grid-computing middleware systems. It facilitates directory/monitoring service, resource scheduling, data replication and security service in its MDS, GRAM, DRS and GSI components respectively. Middleware systems using Globus include Con-

dor [6], Ninf [27], and NetSolve[23]. Condor supports high throughput computing by check-pointing, resuming, and migrating master-slave or parameter-sweep jobs with Unix state-capturing functions. Ninf and NetSolve implements optimized RPC-based computation by dynamically allocating the best-fitted servers to their applications. Another system example, while not using Globus, is Legion that is based on its own object-oriented architecture and currently commercialized as Avaki[15]. In most cases, their target users seem to be those who can submit and manage their jobs from a central server, though some of their resource discovery and scheduling schemes are hierarchical or decentralized.

On the other hand, mobile-agent-based middleware can take advantage of agents' inherent parallelism and navigational autonomy that allows distributed job management. However, as mentioned in Section 1, most systems have not yet departed from the use of a central server. For instance, Catalina assigns to each application an application-delegated manager (ADM) that deploys *task* agents, each managing a process execution at a different machine but returning all process snapshots to ADM [16]. Similarly in J-SEAL2, a client user contacts an *operator*, a job-coordinating server that dispatches his/her job with a *deployment* agent to available *donator* sites where a *mediator* process launches, monitors, suspends, and resumes a user process [3].

Contrary to those, AgentTeamwork aims at maximizing the benefits of mobile agents so as to support decentralized job submission, autonomous job migration, distributed snapshot management, and fault tolerance as allowing various programming models not necessarily restricted to the master-slave model.

5.2 Mobile Agents

Other mobile agent platforms intended for use in a grid computing environment could include IBM Aglets[18], Voyager[22], D'Agents[13], and Ara[21]. We consider four areas of functionality, and explain how UWAgents is distinctive in these areas.

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 a 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, which on the other hand 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. On the other hand, 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. 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 the sentinel and bookkeeper agents involved in its job have terminated. In IBM Aglets, a parent agent can use the *retract* function so as to take all its children back to it, 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, cascading termination or synchronization must be implemented at the user level. In UWAgents, 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 byte-code verification, the allowance model, the currency-based model, and the CORBA security service, they have not focused on agent-to-agent security. On the other hand, UWAgents enforces such security in that agents in different domains are prohibited from communicating with each other using its messaging facility.

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 3, 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 [12].

5.3 Job Scheduling

Job scheduling in grid systems has been well surveyed in [17] and can be categorized into centralized, hierarchical, and decentralized scheduling.

Centralized scheduling eases resource management and co-allocation, while implying disadvantages such as the lack of scalability and fault tolerance. For instance, Condor [6] uses a central *ClassAd* matchmaker that maintains a pool of computing resources. To overcome the disadvantages of centralized scheduling, Condor allows a matchmaker and a user to forward his/her computing request to another matchmaker through the gateway flocking mechanism.

Hierarchical scheduling addresses the scalability and fault-tolerance issues by having high and low level schedulers manage the global and local resources respectively, as retaining the ability of resource co-allocation. AppLes [31] collects resource information from Network Weather Service (NWS) [34] running at each computing node, and dispatches parameter-sweep tasks to lighter nodes, each scheduling their actual execution in local[4]. Other system examples include Legion [29] and 2K [19].

Distributed scheduling has no focal point in resource scheduling and can be, for instance, implemented by a pair of local and meta schedulers at each computing node as in [25]. Jobs are locally submitted to a meta-scheduler that exchanges load information with other nodes, are dispatched to a remote node

[4][17] has classified AppLes in a distributed job scheduler from the organizational view point.

considered of as being light-loaded, and are scheduled by its local scheduler. To alleviate heavy meta-scheduler communication, nearest-neighbor and random-node probing algorithms are used to restrict the number of remote nodes each scheduler can contact [7]. Grid systems based on distributed scheduling include Ninf [27], MOL [11], and Bond [2]. Ninf allows each user to specify a static pool of remote servers to run his/her job. MOL uses MARS (Metacomputer Adaptive Runtime System) that runs at each node to acquire a program's runtime behavior, to monitor network performance, and to migrate jobs to other MARS schedulers [10]. Bond deploys mobile agents that negotiate resource allocation with remote schedulers, based on computational economy or bidding [4].

In AgentTeamwork, each resource agent downloads XML-based resource files from a common ftp server (thus in a centralized scheme), and thereafter repeats probing remote nodes with its child agents independently similar to NWS. Each sentinel agent negotiates job execution with its local computing node as in Bond [2] in a decentralized manner and migrates to one of the unused nodes in its itinerary.

5.4 Snapshot Maintenance

The main concern with snapshot maintenance is where to back up execution snapshots in preparation for future process retrieval. The simplest solution, as used by many systems, maintains all execution snapshots into a primary server or NFS which may suffer from performance bottleneck and low fault tolerance. J-SEAL2 permits each mobile agent to maintain its process snapshot locally, which is however zero fault-tolerant to each process if its local site has been crashed.

The availability of process snapshots can be generally enhanced by active copy and quorum-based protocols [24] that distribute copies of each snapshot to available backup servers. However, when it comes to AgentTeamwork, each sentinel agent needs to keep track of all bookkeeper agents that may even repeat their migration. In addition, any bookkeepers may fail in receiving process snapshots during their migration or suspension, because of which sentinels would need to desperately search every bookkeeper for their latest snapshot.

As a simplified solution, AgentTeamwork sends a process snapshot from each sentinel to its corresponding bookkeeper that further reroutes the snapshot to other two. This scheme avoids performance

bottleneck as in the primary server model, zero fault-tolerance as in J-SEAL2, and exhaustive snapshot search as in the active-copy protocol. The drawback is that a snapshot may not be retrieved if all the three bookkeepers maintaining it have been crashed at once, which we anticipate might rarely happen.

5.5 Job Check-Pointing

The simplest check-pointing is to depend on each application that must specify which data to save. For instance, MPI applications in Legion [29] need to save their own data with *MPI_FT_Save*. A crashed application completely restarts from the top of the *main* function, upon which the application itself must call *MPI_FT_Restore* as directed from *MPI_FT_Init*. DOME [1] and Condor [6] resume a user application from the last checkpoint using Unix state-capturing functions. In other words, they can save an application's program counter as well as their user data.

The major drawback is that these systems do not save in-transit messages which in turn means no recovery from a crash in inter-process communication. (Note that DOME need not even handle it, since all communication takes place as an object call and therefore only objects must be saved.) The Condor MW project has focused on the master-worker model where the MW library in a master process keeps track of and retrieves connections to all its worker processes [5]. However, inter-worker communication is not yet retrievable due to the lack of snapshots taken at each worker.

Rock/Rack has facilitated reliable TCP connections by wrapping actual socket connections and buffering in-transit data in a user address space [35]. Using Rock/Rack, a user process can reestablish a TCP connection to its "migrating" peer process, provided it is informed of this peer's new IP address. Although any communication among migrating processes is technically re-connectible by combining Condor and Rock/Rack, users are forced to develop mobile-aware applications, (which periodically check occurrences of migration and inform a peer process of a new IP address).

AgentTeamwork takes function-based snapshots and resumes an application from the last function call. Similar to Rock/Rack, in-transit messages over TCP connections are logged by the GridTcp library and the user program wrapper. The difference is that our mobile agents take care of automatic TCP reconnections, so that applications are completely relieved from mobile awareness. Unlike Condor-MW,

AgentTeamwork does not restrict applications to the master-slave model for job check-pointing.

6 Conclusions

We have focused on a group of grid-computing users who have no means to share a central cycle server or need to develop applications in a wider range of programming models such as distributed simulation. AgentTeamwork provides those users with a decentralized job execution and fault-tolerant environment in that mobile agents are deployed over Internet to manage a job in their hierarchy. In this paper, we have presented the execution model and implementation techniques of AgentTeamwork. Our analysis demonstrates AgentTeamwork's competitive performance for parallel applications that are characterized by its computational granularity of 40,000 doubles $\times$ 10,000 floating-points, moderate data transfers not combined with both massive and collective communication, and its long execution, (i.e., as three time large as its computational granularity). Finally, our future work will focus on the following three areas:

1. *UWAgents enhancement:* To utilize GridTcp's over-gateway communication feature, we need to enable UWAgents to dispatch agents over different clusters. We must also pay more attention to enhancing UWAgents' security features for the practical use of AgentTeamwork.

2. *Programming support:* We will complete our mpiJava API implementation and design a language preprocessor so that applications are fully described with mpiJava's API and automatically partitioned into a collection of check-pointed functions.

3. *Job scheduling algorithms:* We will put job priority and priority aging in sentinel agents so that, when colliding at the same node, they can negotiate their job execution with each other and decide whether to wait for their turn or migrate to another node if they have a lower priority. Another improvement in AgentTeamwork is to statically allocate future destination nodes to each agent. This scheme will prevent migrating agents from choosing the same destination node from their itinerary, which alleviates agent collisions and negotiations.

We feel that work in these three areas will enhance AgentTeamwork's practicability in grid computing.

Acknowledgments

This research is fully supported by National Science Foundation's Middleware Initiative (No.0438193). We are very grateful to UWB CSS students and graduates Zhiji Huang, Ryan Liu, Enoch Mak, Eric Nelson, Shane Rai, and Duncan Smith, for their programming contributions to the UWAgents, eXist interface, and AgentTeamwork implementation. We also thank Suzanne Schaefer for useful discussions.

References

[1] Jose Nagib Cotrim Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a distributed computing environment. In *Proc. of the 10th International Parallel Processing Symposium – IPPS'96*, pages 218–224, Honolulu, HI, April 1996. IEEE CS.

[2] Ladislau B*ö*l*ö*ni. The bond 3 agent system. White paper, School of Computer Science, University of Central Florida, 2002.

[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] Thomas Casavant and Jon Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transaction on Software Engineering*, Vol.14(No.2):141–154, February 1988.

[5] Condor MW Homepage. http://www.cs.wisc.edu/condor/mw/, 2004.

[6] Condor Project. http://www.cs.wisc.edu/condor/.

[7] Luis Paulo Peixoto do Santos. Load distribution: a survey. Technical report UM/DI/TR/96/03, Department of Informatica, University of Minho, Portugal, October 1996.

[8] Ian Foster and Carl Kesselman, editors. *The Grid 2 Blueprint for a New Computing Infrastructure, 2nd Edition*. Morgan Kaufmann, November 2003.

[9] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, Vol.15(No.3):200–222, January 2001.

[10] Jörn Gehring and Alexander Reinefeld. MARS – a framework for minimizing the job execution time in a metacomputing environment. *Future Generation Comput Systems*, Vol.12(No.1):87–99, 1996.

[11] Jörn Gehring and Achim Streit. Robust resource management for metacomputers. In *Proc. of the 9th IEEE International Symposium on High Performance Distributed Computing – HPDC'00*, pages 105–112, Pittsburgh, PA, August 2000. IEEE-CS.

[12] 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.

[13] 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.

[14] Grid@IFCA commercial grid solutions. http://grid.ifca.unican.es/dissemination/Commercial.htm, 2003.

[15] Andrew S. Grimshaw, Anand Natrajan, Marty A. Humphrey, Michael J. Lewis, Anh Nguyen-Tuong, John F. Karpovich, Mark M. Morgan, and Adam J. Ferrari. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 10, From Legion to Avaki: The Persistence of Vision, pages 265–298. John Wiley & Sons, March 2003.

[16] 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.

[17] 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.

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

[19] Jeferson Roberto Marques, Tomonori Yamane, Roy H. Campbell, and M. Dennis Mickunas. Design, implementation, and performance of an automatic configuration service for distributed component systems. *Software: Practice and Experience*, to appear, 2005.

[20] mpiJava Home Page. http://www.hpjava.org/mpijava.html.

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

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

[23] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. *Grid Computing and New Frontiers of High Performance Processing*, chapter to appear, NetSolve: Grid Enabling Scientific Computing Environments. Elsevior, 2005.

[24] Pradeep K. Shinha. *Distributed Operating Systems: Concepts and Design*, chapter 9.9.7, File Replication, pages 440–447. IEEE CS Press, New York, 1997.

[25] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *Proc. of the 11th International Symposium on High Performance Distributed Computing – HPDC 2002*, pages 359–366, Edinburgh, Scotland, July 2002. IEEE-CS.

[26] Naoya Suzuki. *Research on A Parallel Multi-Agent Simulation System Oriented to Complex Systems*. PhD thesis, University of Tsukuba, Ibaraki 305, Japan, March 2004.

[27] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, implementation and performance evaluation of gridrpc programming middleware for a larg-scale computational grid. In *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Pittsburgh, PA, November 2004.

[28] The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/, 2002.

[29] The Legion Group. Legion 1.8 basic user manual. Technical report, Department of Computer Science, University of Virginia, Charlottesville, VA, 2001.

[30] 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.

[31] Krijn van der Raadt, Yang Yang, and Henri Casanova. Practical divisible load scheduling on grid platforms with APST-DV. In *Proc. of the 19th International Parallel and Distributed Processing Symposium – IPDPS'05*, Denver, CO, April 2005. IEEE CS.

[32] G. Vogt. Delegation of tasks and rights. In *Proc. of the 12th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations & Management – DSOM2001*, pages 327–337, Nance, France, October 2001. INRIA.

[33] C. Wicke, L. Bic, M. Dillencourt, and M. Fukuda. Automatic state capture of self-migrating computations in MESSENGERS. In *Proc. of the 2nd International Workshop on Mobile Agents – MA'98*, pages 68–79. Springer, September 1998.

[34] Rich Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, Vol.30(No.4):41–49, March 2003.

[35] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proc. of the 8th Annual International Conference on Mobile Computing and Networking – MOBICOM'02*, pages 95–106, Atlanta, GA, September 2002. ACM Press.