NSF SCI #0438193: Final Report
Mobile-Agent-Based Middleware for Distributed Job Coordination

Munehiro Fukuda

Computing and Software Systems, University of Washington, Bothell
email: mfukuda@u.washington.edu

March 31, 2008

Abstract

This final report summarizes all the PI's research activities conducted on NSF SCI #0438193: Mobile-Agent-Based Middleware for Distributed Job Coordination. The PI has investigated and evaluated the applicability of mobile agents to grid-computing middleware by implementing and enhancing his AgentTeamwork system that facilitates job deployment, check-pointing, resumption, migration and monitoring in a hierarchy of mobile agents. The system has been implemented in collaboration with the PI's undergraduate research assistants, Ehime University graduate students, and exchange students.

In year 2005, we started our project with installing a 32-node cluster system, and implemented AgentTeamwork's basic job deployment and resumption features based on Java RMI. In year 2006, we extended our computing resources to two clusters, each including 32 nodes, thus bringing up the total to 64, revised the AgentTeamwork system with Java sockets, implemented AgentTeamwork's job coordination, and file transfer over multi clusters. In year 2007, we debugged, enhanced, and documented all AgentTeamwork's components: inter-cluster job deployment and resumption, parallel file distribution and collection, and XML-based resource maintenance and monitoring.

We believe that our main contribution to grid-computing middleware is job coordination in an agent hierarchy that can be considered not only as a self-remapping tree of user processes but also as dynamic file-distribution routes to the most available processor pool. The project outputs are as follows: we published 2 journal papers and 7 conference papers, (all peer-reviewed); gave colloquium presentations at 6 different universities in US and Japan; and made available the AgentTeamwork's software components as the PI's course materials.

Contents

1 Overview

NSF SCI #0438193: Mobile-Agent-Based Middleware for Distributed Job Coordination is an RUI project that investigates and evaluates the applicability of mobile agents to grid-computing middleware by implementing and enhancing the AgentTeamwork system. The system facilitates job deployment, check-pointing, resumption, migration and monitoring in a hierarchy of mobile agents.

The following subsections summarize our achievements for each year from 2005 to 2007.

1.1 Achievement Summary for Year 2005

Year 2005 was the first year of our three-year NSF-granted research activities. Starting with the purchase and installation of research equipments, we achieved the following research work:

1. RMI-based and Java-socket-based implementations of our mobile-agent execution platform (named UWAgents) that serves as AgentTeamwork's infrastructure,
2. the initial implementation of AgentTeamwork's system agents that can dispatch a job, monitor remote computing resources, and detect agent crashes in their hierarchy,
3. a study of AgentTeamwork's preprocessor and code cryptography that translates a java program to the code accepted by AgentTeamwork and encrypted for security purposes, and
4. an implementation of the mpiJava API with Java sockets and our fault-tolerant socket library named GridTcp.

1.2 Achievement Summary for Year 2006

Year 2006 brought in the following achievements as the mid year of our project.

1. an enhancement of UWAgents mobile-agent execution platform so as to deploy agents into cluster-private nodes,
2. an implementation of AgentTeamwork's inter-cluster job deployment in an agent hierarchy,
3. an implementation of AgentTeamwork's parallel and pipelined file-transfer mechanism as well as fault-tolerant file library named GridFile,
4. an implementation of AgentTeamwork's runtime remote-resource monitoring and database management, and
5. Java application development

1.3 Achievement Summary for Year 2007

1. an enhancement and completion of AgentTeamwork's inter-cluster job deployment and resumption
2. an enhancement and completion of AgentTeamwork's parallel file distribution, collection, and consistence maintenance over remote computing nodes, all made available through GridFile and RandomAccessFile libraries
3. a completion of AgentTeamwork's XML resource database and GUI
4. a continuation of Java application development: matrix multiplication, Schroedinger's wave simulation, Mandelbrot, distributed grep, and two-dimensional molecular dynamics
5. a documentation of AgentTeamwork's software components: a user manual, a programming manual, and a design sheet.

In addition to the research activities and major finding, this final report will give details of the PI's student supervision, publications and budget activities as well as his post-award research plan.

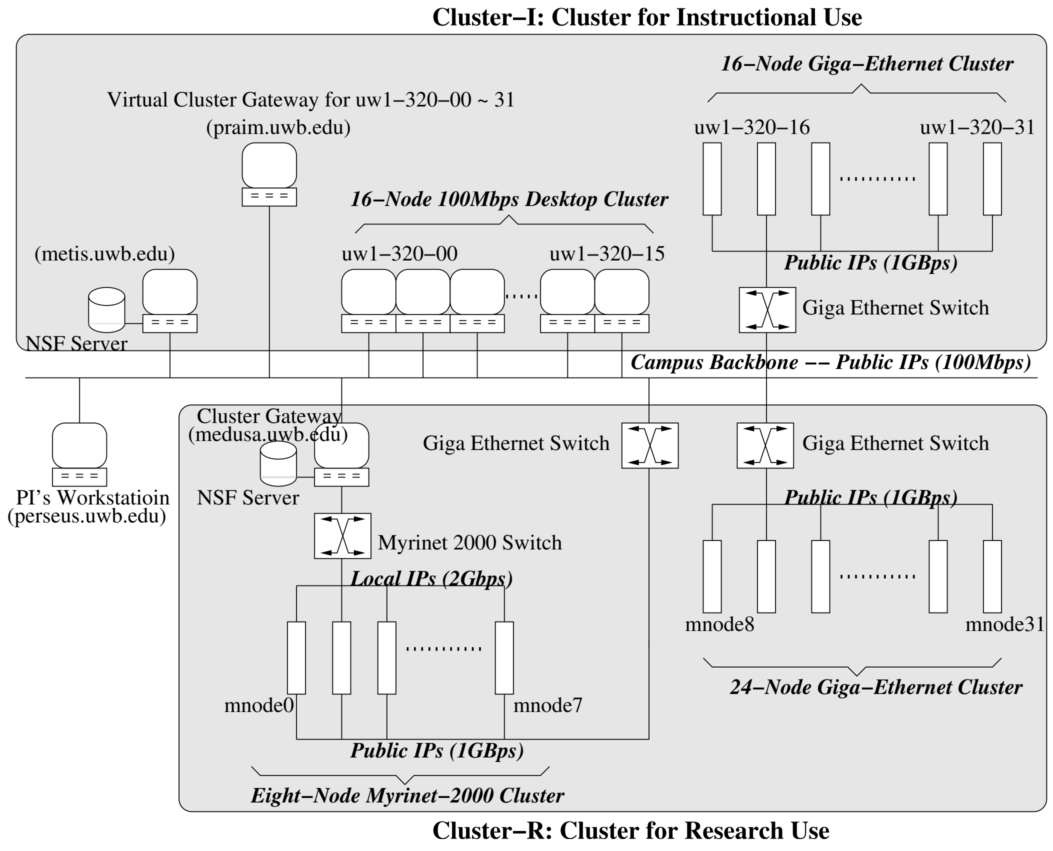


Figure 1: A Configuration of Research Equipments

2 Research Activities

This section describes a configuration of our research equipments, an overview of the AgentTeamwork system, and our implementation of each system component.

2.1 Research Equipments

We used two clusters for implementing the AgentTeamwork system and evaluating its job-coordination performance. Figure 1 and Table 1 summarize their configuration and specification. Cluster-R is the one the PI has purchased with his research funds including this NSF award, whereas Cluster-I is the departmental cluster mainly used for instructions and upgraded every three through to five years.

AgentTeamwork assumes that clusters nodes are located in a private IP domain and thus only accessible from their cluster gateway. For Cluster-R, we used the *medusa* NFS server as the gateway. For Cluster-I, we used the *priam* desktop machine as the gateway so as not to deteriorate the performance of the instructional *metis* NFS server.

2.2 System Overview

AgentTeamwork allows a new computing node to join the system by running a UWAgents mobile-agent execution daemon to exchange agents with others [8]. UWAgents maintains a parent-child relationship among mobile agents in a hierarchy that behaves similar to a process tree in operating systems but dynamically extends over network. Using this feature, AgentTeamwork deploys a user job to remote

Cluster-R: a 32-node single-core cluster

Gateway node:		
	specification	**outbound**
	1.8GHz Xeon x2, 512MB memory, 70GB HD, and SunNFS 4.1.3 installed	100Mbps
Computing nodes:		
#nodes	**specification**	**inbound**
8	2.8GHz Xeon, 512MB memory, and 60GB HD	2Gbps
24	3.2GHz Xeon, 512MB memory, and 36GB HD	1Gbps

Cluster-I: a 32-node dual-core cluster

Gateway node:		
	specification	**outbound**
	1.5GHz Xeon, 512MB memory, 40GB HD, and SunNFS 4.1.3 installed	100Mbps
Computing nodes:		
#nodes	**specification**	**inbound**
16	2.13GHz Intel Core2, 1GB memory, and 40GB HD	100Mbps
16	1.8GHz Dual-CoreAMD Opteron, 1GB memory, and 40GB HD	1Gbps

Table 1: Cluster specifications

computing nodes with a several types of agents in a hierarchy. They are distinguished as commander, resource, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, job deployment and monitoring, and job-execution bookkeeping respectively.

Figures 2 and 3 illustrate a job deployment in an agent hierarchy and its execution through agent interaction respectively.

First, a user submits a new job with a commander agent through AgentTeamwork's GUI (named *SubmitGUI*) or using some command line utilities. The commander agent migrates to a given XML resource database (named *XDBase*) and spawns a resource agent (*id 1* in Figure 3) that collects new computing-node information from a shared ftp server, registers it to the local XDBase, and retrieves a list of remote machines fitted to the job execution, (named a *resource itinerary*) [7].

The commander thereafter spawns a pair of agents, a sentinel with *id 2* and a bookkeeper with *id 3*, each hierarchically deploying as many children as requested in the resource itinerary. (Figure 3 shows only one child sentinel with *id 9* for simplicity.) If these computing nodes reside over multiple clusters, agents are deployed to a different cluster head or gateway where they further deploy a hierarchy of children to cluster-internal nodes [3]. Figure 2 shows an example where 11 child sentinels are deployed over clusters 0 and 1.

Before starting a user application, the commander, sentinels and bookkeepers exchange their IP address information through their hierarchy (with *talk(descendant_location)* and *talk(all_locations)* in Figure 3). Thereafter, each sentinel starts a user program wrapper with a unique MPI rank, (rank 0 through to 10 in the example) at a different machine. Launched from the wrapper, a user process periodically takes its computation snapshot and orders its local sentinel agent to send the snapshot to the corresponding bookkeeper (with *sendSnapshot( )* and *talk(save_snapshot)*). At every checkpoint all the user processes are automatically barrier-synchronized (with *tcp.commit( )*) so as to prevent any process from advancing too fast and thus from saving too many old messages in a snapshot. Each sentinel agent exchanges a ping and an acknowledgment with its children for error detection, and resumes them upon a crash. A bookkeeper maintains and retrieves the corresponding sentinel's snapshot on demand.

With *initSocket( )* and *receiveGUIMsg( )*, the commander establishes a socket to SubmitGUI and keeps receiving input files as well as the standard input in smaller packets. Repetitive calls of *talk(grid_infile)* relay these packets as inter-agent messages from the commander down to all user processes through their agent hierarchy. On the other hand, *talk(grid_outfile)* calls return output files and the standard output in a reversed direction from each sentinel up to the commander agent.

Finally, a job termination is agreed among all agents with *talk(end_user_program)* and notified to

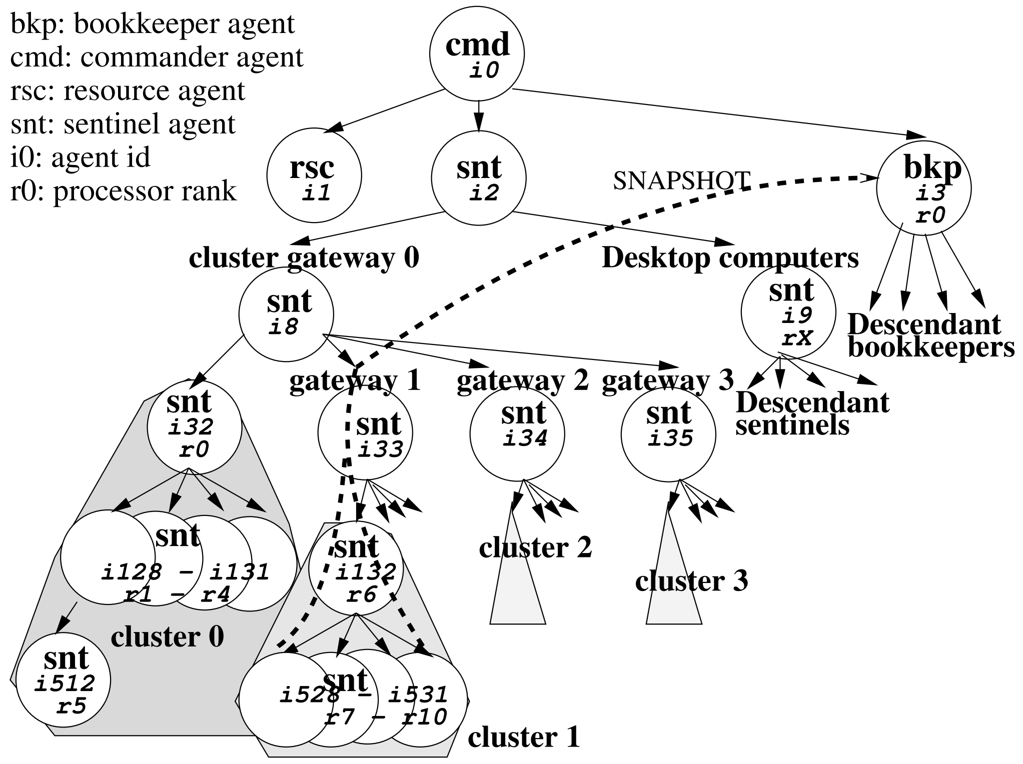


Figure 2: Job deployment by AgentTeamwork

SubmitGUI with *sendMsg(end_user)*, upon which they are terminated with *talk(kill_agent)*.

2.3 Programming Model

Figure 4 shows the AgentTeamwork execution layers from the top application level to the underlying operating systems. (Note that all components written in bold are AgentTeamwork's components and that all those in a shaded box are instantiated from a sentinel agent at each remote node.) The system facilitates inter-process communication with mpiJava [10] as well as Java Sockets for Java-based parallel applications. It also supports Java FileInputStream, FileOutputStream, and RandomAccessFile classes for remote file accesses. Since mpiJava has no MPI-I/O support, we have implemented the same concept in SubmitGUI and RandomAccessFile. The former assists a user in defining his/her file stripes with its GUI menus, whereas the latter allows an application program to access different file stripes with the *seek* method.

All these Java-based packages (including mpiJava) were re-implemented using *GridTcp* and *GridFile* in that we have developed fault-tolerant TCP and file manipulation. GridTcp monitors TCP links emanating from its local process, maintains the history of in-transit TCP messages, and restores all broken communication upon a process resumption. It also supports multi-cluster communication as in MPICH-G2 and tolerates cluster crashes [4]. GridFile implements an interface between a user program and AgentTeamwork by maintaining file contents in its error-recoverable queues.

Below these components is Ateam that provides a user application with check-pointing methods to explicitly take its computation snapshots through serialization of the application itself, GridTcp, and GridFile, all referenced to from the Ateam object. Ateam is then wrapped with a user program wrapper, one of the threads running within a sentinel agent. Recovered from a crash, the sentinel agent restarts its user program wrapper that resumes all the wrapped components from the latest snapshot.

The sentinel and the other AgentTeamwork's agents are executed on top of the UWAgents mobile

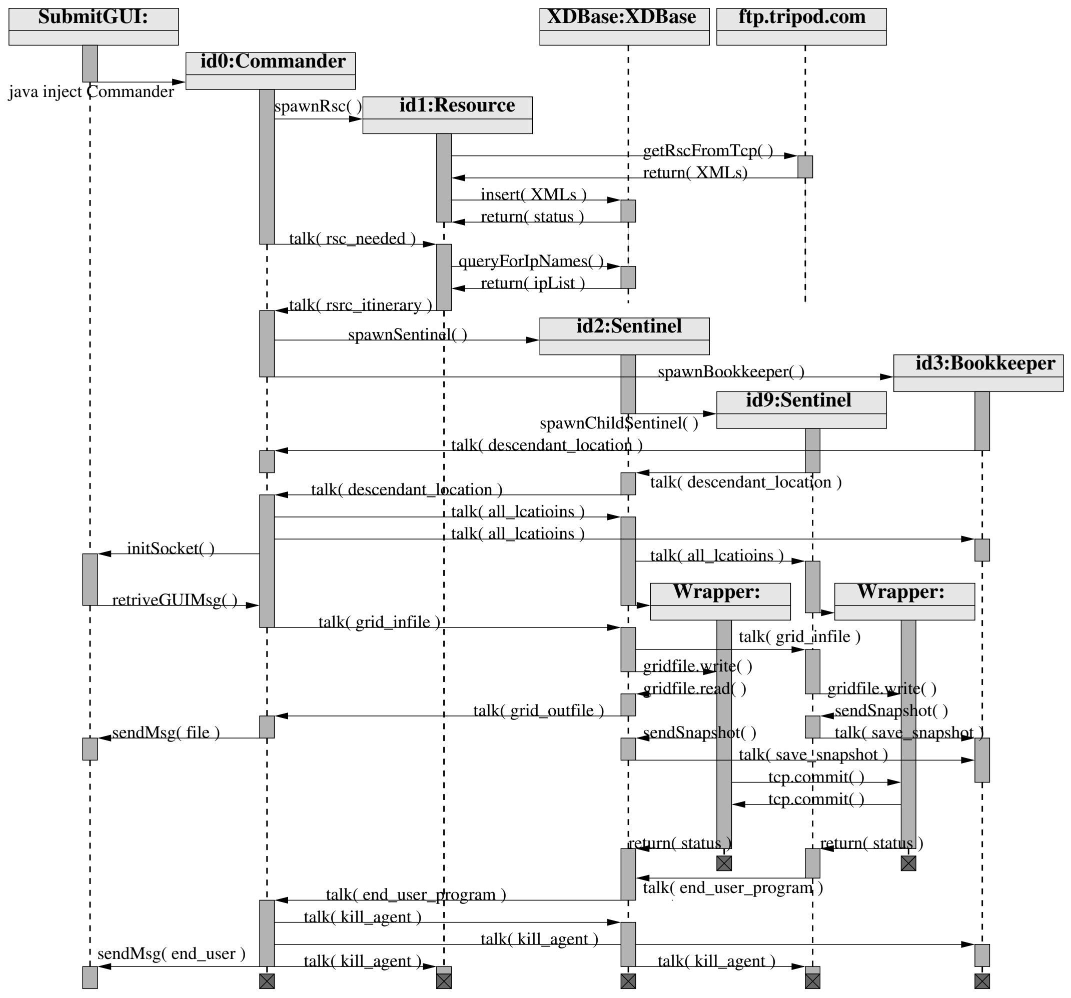


Figure 3: Job coordination through agent interaction

agent execution platform which we have developed with Java as an infrastructure for agent-based grid computing.

Figure 5 shows a Java application executed on and check-pointed by AgentTeamwork. Besides all its serializable data members (lines 3-4), the application can register local variables to save in execution snapshots (lines 38-39) as well as retrieve their contents from the latest snapshot (lines 32-33). At any point of time in its computation (lines 12-28), the application can take an on-going execution snapshot that is serialized and sent to a bookkeeper agent automatically (lines 22 and 26). As mentioned above, it can also use Java-supported files and mpiJava classes whose objects are captured in snapshots as well (lines 14, 16, and 24).

Focusing on RandomAccessFile, a file can be shared among all processes, while each stripe is actually allocated to and thus owned by a different process. Figure 5 assumes that each process owns and thus writes its $rank$ to a one-byte stripe (lines 17-18) that is read by another process with $rank - 1$ upon a barrier synchronization (lines 19-21).

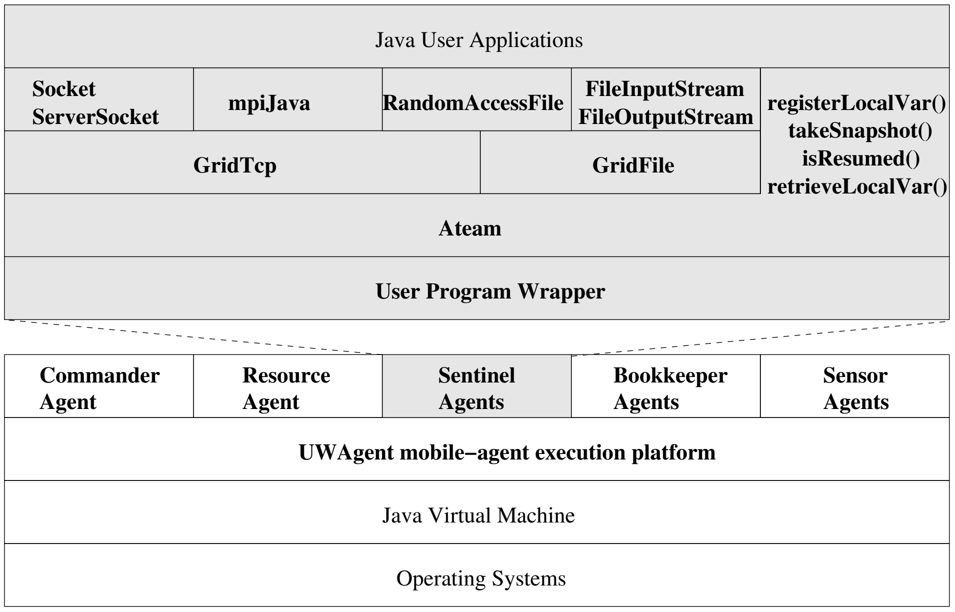


Figure 4: AgentTeamwork execution layer

2.4 Implementation

In the following, we explain our implementation of the system features: (1) UWAgents mobile-agent execution platform; (2) job deployment by commander, sentinel, and bookkeeper agents; (3) parallel file transfer in an agent hierarchy; (4) resource management and monitoring with an XML database, a resource agent, and sensor agents; (5) Job injection and file partitioning with GUI; and (6) AgentTeamwork applications.

2.4.1 UWAgents Mobile-Agent Execution Platform

UWAgents is a Java-based mobile-agent execution platform that runs at each computing node to exchange AgentTeamwork's agents with other nodes. We implemented UWAgents with Java RMI, thereafter re-implemented it with Java sockets for better performance, and finally enhanced it so as to deploy agents over multi-clusters.

Figure 6 sketches that a new agent is submitted from a Unix shell prompt to form an agent domain where it can recursively fork offspring as the domain root with *id* 0. 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 6, 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. Since AgentTeamwork submits a new job with a commander agent, (i.e., a root agent with id 0), the job will be executed inside this commander's agent domain, thus without being interfered by other jobs.

Agents can travel between two separate networks or clusters if those networks are linked by one or more gateway machines, provided each machine runs UWPlace. Agents on one network can then send messages to those on the other network. The following code fragment shows an example of agent migration through two cluster gateways, (i.e., *gateway_1* and *gateway_2*) to a computing node named *node1* in a private address domain:

```
String[] gateways = new String[2];
```

```
import AgentTeamwork.Ateam.*;
public class MyApplication extends AteamProg {
  private int phase;                          // snapshot phase
  private RandomAccessFile raf;               // RandomAccessFile
  public MyApplication(Object o){}            // the system-reserved constructor
  public MyApplication( ) {                   // a user-own constructor
    phase = 0;
  }
  private boolean userRecovery( ) {
    phase = ateam.getSnapshotId( );           // version check
  }
  private void compute( ) {                   // user computation
    ...;
    raf = new RandomAccessFile(               // create a RandomAccessFile object
              new File("infile"), "rw" );
    int rank = MPI.COMM_WORLD.Rank( );
    raf.seek( rank );                         // go to my stripe
    raf.write( rank );                        // write my rank
    raf.barrier( );                           // synchronize with other ranks
    int[] data = new int[1];                  // prepare a variable to store data
    int data[0] = raf.read( );                // read my rank + 1
    ateam.takeSnapshot( phase++ );            // check-point intermediate computation
    raf.close( );                             // close the RandomAccessFile object
    MPI.COMM_WORLD.Reduce( data, 0,           // an MPI function
            data, 0, 1, MPI.INT, MPI.Sum, 0 );// sum up what each rank has read
    ateam.takeSnapshot( phase++ );            // check-point intermediate computation
    ...;
  }
  public static void main( String[] args ) {  // start a user program
    MyApplication program = null;
    if ( ateam.isResumed( ) ) {               // the program has resumed.
      program = (MyApplication)               // retrieve the latest snapshot
         ateam.retrieveLocalVar( "program" );
      program.userRecovery( );
    } else {                                  // program has invoked from its beginning.
      MPI.Init( args );                       // invoke mpiJava
      program = new MyApplication( );         // create an application
      ateam.registerLocalVar( "program",      // register my program
                              program );
    }
    program.compute( );                       // now go to computation
    MPI.Finalize( args );                     // end mpiJava
} }
```

Figure 5: File operations in AgentTeamwork's application

```
gateways[0] = "gateway_1";
gateways[1] = "gateway_2";
hop("node1", gateways, "nextFunction", args );
```

UWAgents was also enhanced to have an agent temporarily cache its communication counterpart's IP address after their first communication as far as they reside in the same network domain. An agent can then send messages to its final destination or its gateway agent rather than relay them through the ascendant and/or descendant agents in the hierarchy. This new feature allows AgentTeamwork to deliver repetitive snapshots from a sentinel agent straightly to its bookkeeper.

2.4.2 Intra- and Inter-Cluster Job Deployment

Figure 7 shows AgentTeamwork's hierarchical job deployment in a single address domain, which brings the following four benefits: (1) a job is deployed in a logarithmic order; (2) each agent computes its identifier and corresponding MPI rank without a central name server; (3) each sentinel monitors its parent and child agents as well as resumes them upon a crash in parallel; and (4) each user process is check-pointed by a different sentinel and its snapshot is maintained by a separate bookkeeper, which improves snapshot availability and thus enhances fault tolerance.

Despite those merits, this algorithm cannot be re-used directly for inter-cluster job coordination. The major problem is how to divide an agent hierarchy into subtrees, each allocated to a different cluster.

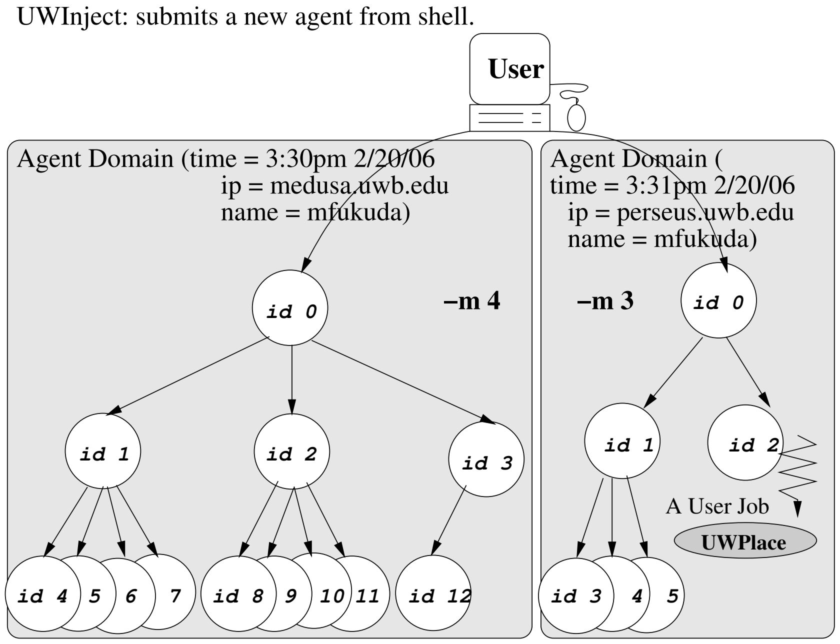


Figure 6: Agent domain

Without subtree generation, an agent hierarchy would be deployed over multiple clusters in a pathetic manner where a sentinel agent in a cluster-private domain might need to monitor its parent and children, some residing at a different cluster. This burdens a cluster gateway with rerouting all ping messages from such a cluster-internal agent to different clusters.

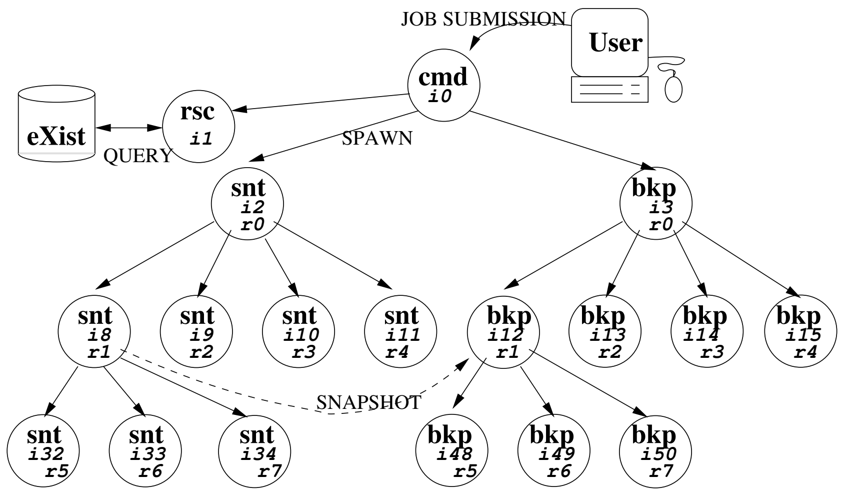
We addressed this multi-cluster problem by forming a tree of sentinel agents, each residing on a different cluster gateway and further generating a subtree from it that covers all its cluster-internal nodes. Figure 2 in Section 2.2 describes the final implementation of our inter-cluster job deployment. It distinguishes clusters with a private IP domain from desktops with a public IP address by grouping them in the left and right subtrees respectively of the top sentinel agent with *id* 2. In the left subtree, all but the leftmost agents at each level are deployed to a different cluster gateway. Note that we call them *gateway agents* in the following discussions and consider the left subtree's root with *id* 8 as the first gateway agent. Each leftmost agent and all its descendants are dispatched to computing nodes below the gateway managed by this leftmost agent's parent. We distinguish them from gateway agents as *computing-node agents*.

This deployment has three merits: (1) all gateway agents are managed in a tree of non-leftmost nodes and simply differentiated from computing-node agents; (2) each gateway agent can calculate its position in the left subtree starting from agent 8 so as to locate a cluster it should manage; and (3) each computing-node agent can calculate its position within a subtree starting from its gateway agent, (i.e., within its cluster) so as to locate a computing node it should reside and to calculate its MPI rank from its agent id.

2.4.3 File Distribution and Collection

File partitioning in AgentTeamwork is based on the MPI-I/O concept [9]. A user is supposed to instruct the system through its GUI, (named *SubmitGUI*) how to partition a given file into stripes and to allocate each to a different remote process by specifying the corresponding rank's *file view*, namely a repetition of an identical *filetype* that is tiled with multiple *etype*s and *holes*.

Upon generating file stripes, SubmitGUI launches a commander agent that reads those strides, de-



bkp: bookkeeper agent	i0: agent id
cmd: commander agent	r0: processor rank
rsc: resource agent	
snt: sentinel agent	

Figure 7: AgentTeamwork's original algorithm for job coordination

ploys a user job to remote sites through a hierarchy of sentinel agents, and thereafter keeps delivering each file stride through this hierarchy. Figure 8 illustrates this stride distribution. Given a hierarchy-unique identifier (abbreviated as an *id*), each sentinel determines an MPI rank for its user process, discovers all *id*s of its descendant agents, and similarly calculates their MPI ranks. With this knowledge, a sentinel agent can pass a file stride from its parent to the child agent whose descendant subtree includes this stride's final destination. To reduce inter-agent communication overhead, a sentinel aggregates and passes multiple strides at once to the same child agent.

File collection starts as soon as a user process writes data to files. It uses an agent hierarchy in similar to file distribution while its direction is reversed from each sentinel to the commander agent. From its nature, the closer to the commander a sentinel agent is located, the busier traffic of file messages it is exposed to.

To alleviate this traffic congestion at higher-level agents, we have implemented two file-collection paths in their hierarchy as shown in Figure 9. One is an ordinary path to transfer file messages from a child to its parent agent. The other is a bypass (drawn thick in Figure 9) from a child to its grandparent as skipping over its parent agent. This bypass allows each agent to receive file messages from four of its grandchildren ahead, thus relieves it from being stuck with accepting a bulk of messages from its children at once, furthermore alleviates the waste of memory space to spool in-transit messages, and therefore prevents unnecessary paging-out operations.

We have also implemented the GridFile class that is instantiated at each sentinel (and thus running under its corresponding user process) so as to wrap all files exchanged with the user process. Those files are maintained in local memory, captured in an execution snapshot, and de-serialized at a new site when the corresponding user process resumes its computation after a migration or a crash.

2.4.4 Resource Database and Monitoring

Spawned from a commander agent, the resource agent downloads new XML resource specifications from a shared ftp server, maintains them in its local XML database for both initial and runtime resource status, and updates XML files with runtime status that has been reported from sensor agents. These features have allowed the resource agents to choose a runtime-based collection of remote IP names

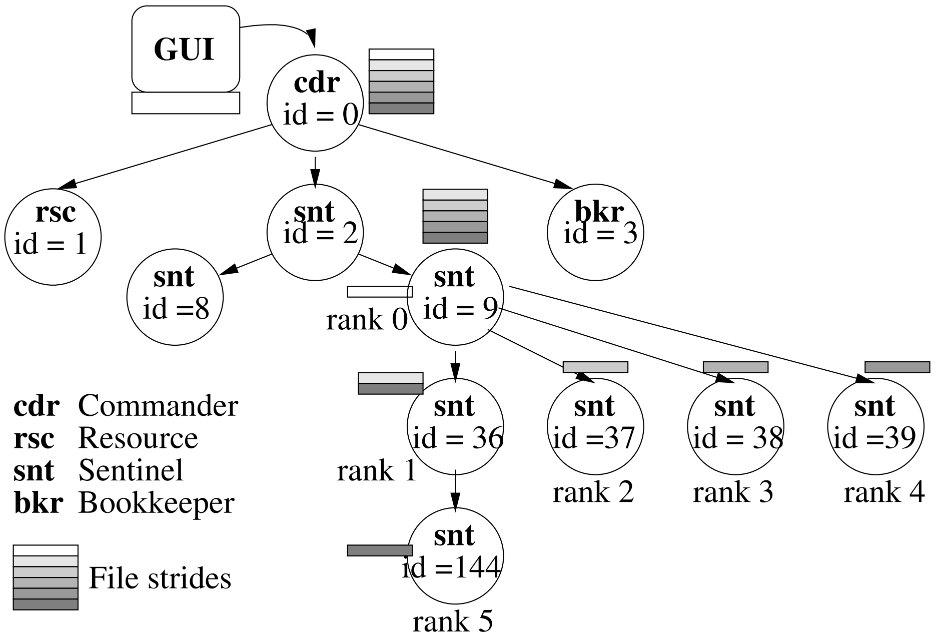


Figure 8: File distribution from the commander in an agent hierarchy (for file read)

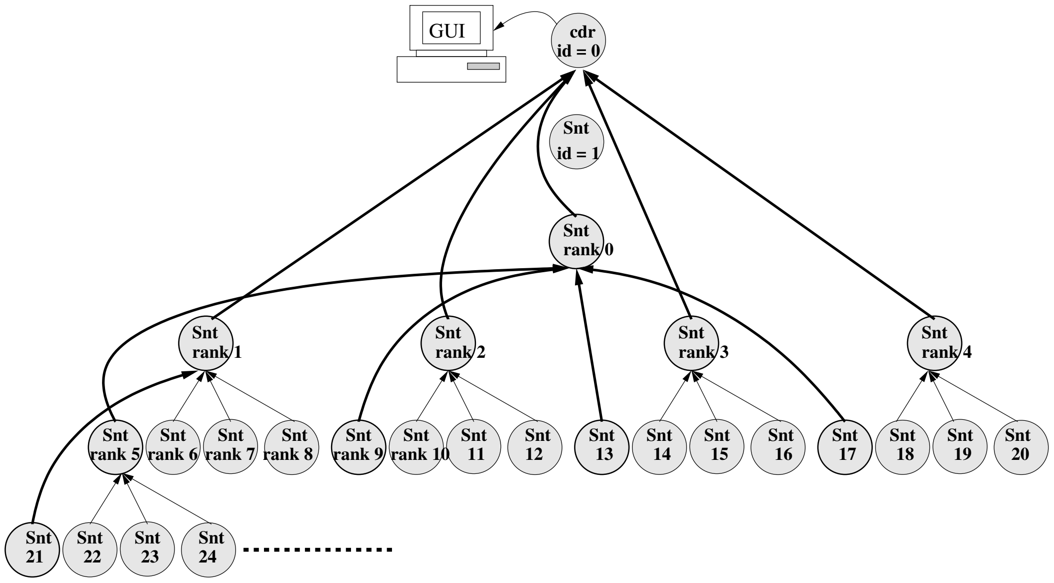


Figure 9: File collection to the commander in an agent hierarchy (for file write)

fitted to a given job execution.

For the purpose of facilitating dynamic job-scheduling in the database, we have implemented our own XML database instead of using the eXist open software that we used at our early development stage:

1. **XDBase.java** is our database manager that maintains two collections of XML-described resource files in a DOM format: one maintains initial XML files downloaded from a shared ftp server and the other is specialized to XML files updated at run time for their latest resource information. Upon a boot, *XDBase.java* waits for new service requests to arrive through a socket.
2. **Service.java** facilitates basic service interface to *XDBase.java* in terms of database management, query, deletion, retrieval, and storage, each actually implemented in a different sub class. *Service.java* receives a specific request from a user either through a graphics user interface or a resource agent, and passes the request to *XDBase.java* through a socket.
3. **XDBaseGUI.java** is an applet-based graphics user interface that passes user requests to *XDBase.java* through *Service.java*.
4. **XCollection.java** is a collection of sub classes derived from *Service.java*, each implementing a different service interface. A resource agent carries *XCollection.java* with it for the purpose of accessing its local *XDBase.java*.

A pair of sensor agents, each deployed to a different node, periodically measure the usage of CPU, memory, and disk space specific to their node as well as their peer-to-peer network bandwidth. These two agents are distinguished as a client and a server sensor. The client initiates and the server responds to a bandwidth measurement. They spawn child clients and servers respectively, each further dispatched to a different node and forming a pair with its correspondence so as to monitor their communication and local resources.

The sensor agents' inter-cluster deployment is similar to that of sentinel agents, while sensors must form pairs of a client and a server. Upon an initialization, the resource agent takes charge of spawning two pairs of root client and server sensors, one dedicated to desktop computers and the other deployed to clusters. The former pair recursively creates child clients and servers at different desktops in the public network domain. The latter pair migrate to different cluster gateways where they creates up to four children. Two of them migrate beyond the gateway to cluster nodes as further creating offspring. The other two are deployed to different cluster gateways, and subsequently repeat dispatching offspring to their cluster nodes and other cluster gateways.

With this hierarchy, the resource information of all cluster nodes is gathered at their gateway and thereafter relayed up to the resource agent that eventually reflects it to the local XML database.

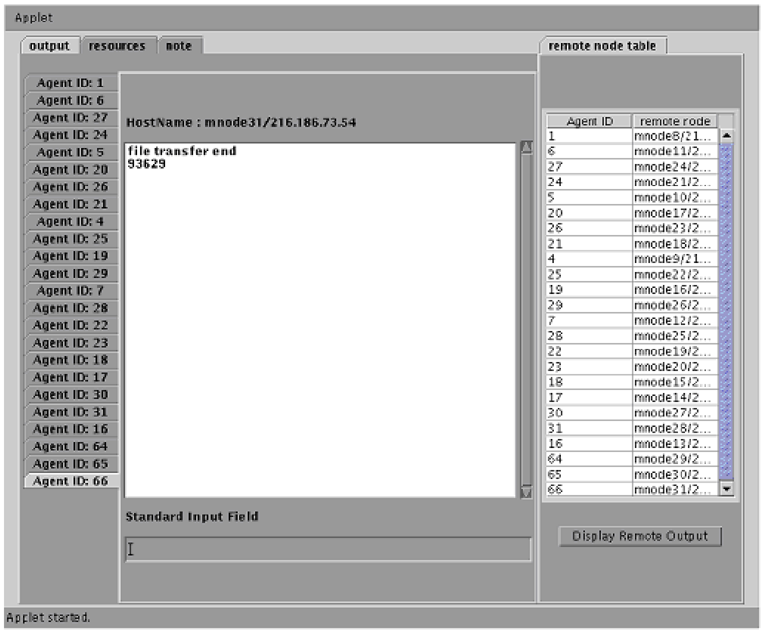
2.4.5 Graphics User Interface

Figure 10 shows our *submitGUI* job-submission interface that allows users to submit their job through the graphical menu, to specify their data files handled in their application, to type in keyboard inputs at run time, and to view the status of and the standard output of each sentinel agents.

A user can also define MPI-I/O-based *filetype*s through SubmitGUI. Assume four different *filetype*s as shown in Figure 11-A, where each *filetype* allocates the $(i \times 2)th$ and $(i \times 2 + 1)th$ *etype*s (as a *float* and a *double* respectively) to rank i. Based on this definition, Figure 11-B captures a snapshot of SubmitGUI's input menu that makes the $0th$ and $1st$ *etype*s visible to rank 0.

2.4.6 Applications

Besides the Java Grande MPJ benchmark programs [14], we have ported the following five programs to mpiJava [10] and thereafter to AgentTeamwork:



```
HostName : mnode31/216.186.73.54
file transfer end
93629
```

Agent ID	remote node
1	mnode8/21...
6	mnode11/2...
27	mnode24/2...
24	mnode21/2...
5	mnode10/2...
20	mnode17/2...
26	mnode23/2...
21	mnode18/2...
4	mnode9/21...
25	mnode22/2...
19	mnode16/2...
29	mnode26/2...
7	mnode12/2...
28	mnode25/2...
22	mnode19/2...
23	mnode20/2...
18	mnode15/2...
17	mnode14/2...
30	mnode27/2...
31	mnode28/2...
16	mnode13/2...
64	mnode29/2...
65	mnode30/2...
66	mnode31/2...

```
Applet started.
```

Figure 10: SubmitGUI's menu example

Programs	**Descriptions**	**Major communication**
Wave2D.java	2D Schroedinger Equation	SendRecv with neighboring ranks
MD.java	2D Molecular Dynamics	Bcase or alternatively SendRecv with neighbors
Mandel.java	Mandelbrot	Master-slave communication with Bcast, Send, and Recv
DistributedGrep.java	Distributed Grep	Master-slave communication with Send and Recv
MatrixMult.java	Matrix Multiplication	Master-slave communication with Send and Recv

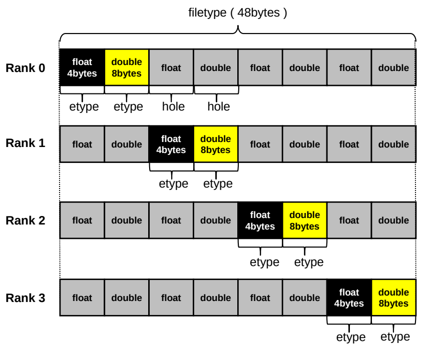
One of our pursuits in AgentTeamwork is to allow users to develop their applications based on not only the master-slave model but also other communication models. For instance, Wave2D passes a new wave amplitude value from one to another neighboring cell, which is implemented in heart-beat communication between two adjusting ranks. MD can be coded in a similar manner by restricting each molecular not to travel beyond a collection of cells allocated to one computing node, (i.e., not to jump over a neighboring node).

To increase AgentTeamwork's applications, the PI has also designed an mpiJava-based programming assignment for his CSS434 "Parallel and Distributed Programming" course where his students parallelize scientific applications using mpiJava.

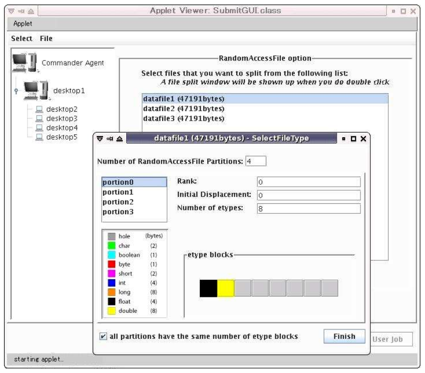
3 Major Findings

Through this three-year research project, we investigated the following five aspects of the AgentTeamwork system: (1) AgentTeamwork-suitable computational granularity, (2) computationally-scalable applications for AgentTeamwork, (3) the efficiency of our multi-cluster job-deployment algorithm, (4) the performance of file transfer in an agent hierarchy, and (5) job check-pointing overheads.

Items 1 and 2 were introduced in our annual report for year 2005 [1] and detailed in our International Journal of Applied Intelligence paper [5]. Item 3 was introduced in the annual report for year 2006 [2] and explained in our GCA'07 conference paper [3]. Items 4 and 5 were investigated in year 2007 and described in our Journal of Supercomputing paper [6]. The following subsections summarize these five items.



(A) An example of filetypes


Applet Viewer: SubmitGUI.class

Applet

Select File

Commander Agent
desktop1
desktop2
desktop3
desktop4
desktop5

RandomAccessFile option

Select files that you want to split from the following list:
A file split window will be shown up when you do double click

datafile1 (47191bytes)
datafile2 (47191bytes)
datafile3 (47191bytes)

datafile1 (47191bytes) - SelectFileType

Number of RandomAccessFile Partitions: 4

portion0
portion1
portion2
portion3

Rank: 0
Initial Displacement: 0
Number of etypes: 8

		(bytes)
▨	hole	
🟩	char	(2)
🟦	boolean	(1)
🟥	byte	(1)
🟪	short	(2)
🟦	int	(4)
🟧	long	(8)
⬛	float	(4)
🟨	double	(8)

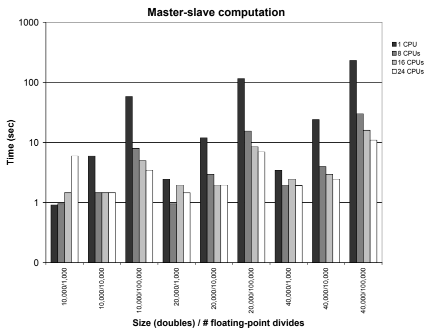
etype blocks

[✓] all partitions have the same number of etype blocks

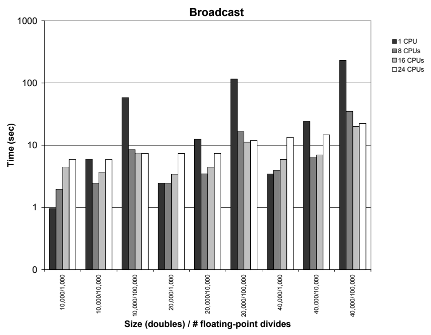
starting applet..

(B) submitGUI

Figure 11: MPI-IO-based file partitioning



(A) Master slave



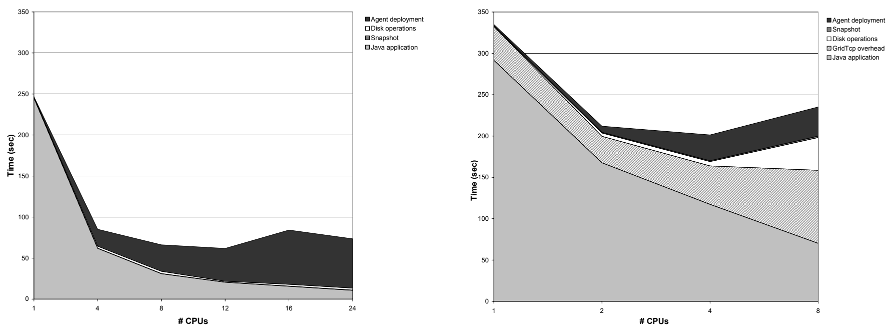
(B) Heartbeat

Figure 12: Computational granularity

3.1 Computational Granularity

Figure 12 shows mpiJava-A's computational granularity when it has executed our *MasterSlave* and *Broadcast* test programs. Both repeat a set of floating-point computations followed by inter-processor communication and an execution snapshot. The 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. The communication pattern of MasterSlave is data exchange between the master and each slave node, whereas that of Broadcast is to let each node broadcast its entire data set to all the other nodes. In other words, MasterSlave involves the lightest communication, while Broadcast incurs the heaviest communication overhead.

MasterSlave has demonstrated its better parallelism beyond 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. We have also coded and run a Heartbeat program where each node exchanges their local data with its left and right neighbors. It has demonstrated computational granularity similar to MasterSlave.



(A) Series (B) MolDyn

Figure 13: Computational Scalability

3.2 Computational Scalability

Figure 13 shows AgentTeamwork's computational scalability and overhead factors when it has executed two Java Grande MPJ Benchmark programs: (1) *Series* that computes $40,032/\ \#nodes$ Fourier coefficients at a different processor and collects all results at the master node, and (2) *MolDyn* that simulates molecular dynamics of 8,788 particles ($8,788 \times 9 = 79,092$ doubles) and exchanges the entire spatial information among processors every simulation cycle. Needless to say, Series and MolDyn represent our MasterSlave and Broadcast granularity test programs respectively.

As shown in Figure 13-A, Series itself is scalable for the number of computing nodes. The largest overhead was agent deployment whose elapsed time was however upper-bounded in logarithmic due to our hierarchical deployment algorithm.

On the other hand, as shown in Figure 13-B, MolDyn has exhibited more overhead in its communication and snapshot-saving operations than agent deployment, which can be characterized in its broadcast communication. This benchmark program 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.

3.3 Job Deployment over Multi-Clusters

For this evaluation, we have used a master-worker test program that does nothing rather than simply exchanges a message between rank 0 and each of the other ranks. Our evaluation has considered the following two scenarios of job deployment and termination as increasing the number of computing nodes engaged in a job: (1) *depth first*: uses up *cluster-R*'s computing nodes first for a submitted job, and thereafter allocates *cluster-I*'s nodes to the same job if necessary. (2) *breath first*: allocates both *cluster-R*'s and *cluster-I*'s computing nodes evenly to a job. For these two scenarios, we have compared AgentTeamwork with Globus that delegates the corresponding MPICH-G2 test program to these two clusters, each mandated by OpenPBS.

Figure 14 compares their performance. For the *depth first* scenario, AgentTeamwork always performed faster than Globus/OpenPBS, however both systems increased their job-deployment overhead sharply from 48 to 64 computing nodes. This is because the test program made a half of worker processes communicate with the master on the other cluster. For *breath first*, Globus/OpenPBS fluctuated its performance till 32 nodes while increasing more eminent overheads due to its linear job deployment over multiple clusters. On the other hand, AgentTeamwork showed its logarithmic increase of job-deployment overhead.

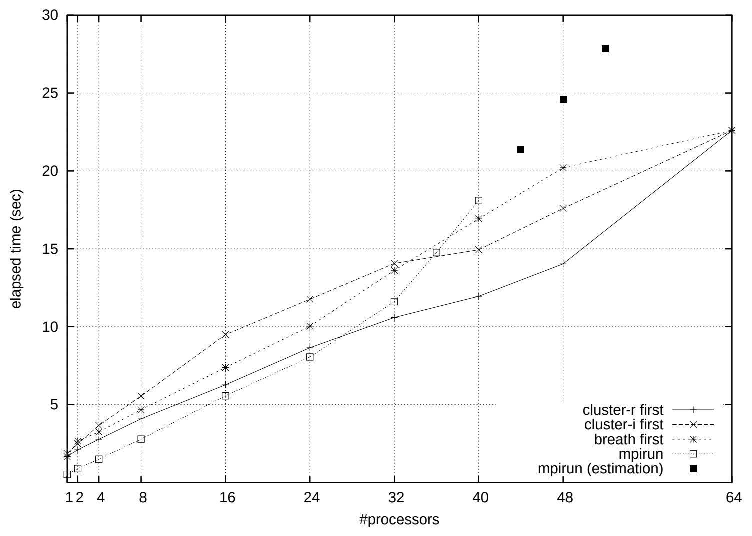


Figure 14: Job deployment effect by cluster node allocation.

3.4 File Transfer in an Agent Hierarchy

We measured file duplication performance by having each of 64 computing nodes read the same file whose size varies from 8M to 256M bytes. We have compared the following three systems:

1. **AgentTeamwork's file duplication**: Injected from the Unix shell, a commander agent accesses its local disk and reads a given file every 16MB into file messages. Those file messages are then forwarded, duplicated, and delivered to 64 sentinels through their agent hierarchy. Upon receiving all messages, each sentinel acknowledges to the commander.
2. **Sun NFS**: We have coded the corresponding Java program that makes all 64 processes read the same file with a 16MB basis through SunNFS Version 4.1.3 and send a "completion" signal to their master process.
3. **User-Level Collective I/O**: To mitigate disk accesses at a user level, we have also revised the above NFS version using collective I/O [13] where the master process reads every 16MB block of a given file and immediately broadcasts it through Java sockets to all the slaves until reading up the file.

We also measured the performance of AgentTeamwork's random-access file transfer measured in both file read and write operations, namely file distribution to and collection from remote processes.

4. **AgentTeamwork's file-stripe distribution**: A random-access file has been partitioned into 64 file stripes *a priori*. We have then measured the total time elapsed for the entire sequence where 64 file stripes are read into a commander agent, relayed in parallel through an agent hierarchy, repeatedly aggregated or fragmented into 16MB file messages, and finally delivered to a different sentinel agent. (Note that this file-stripe transfer involves apparently no message duplication.)
5. **AgentTeamwork's File-stripe collection**: We first distribute a different file stripe to each of 64 sentinels, synchronize all the sentinels, and start a timer at the commander when receiving a signal from the sentinel with rank 0. We have then measured the total time elapsed for the entire sequence

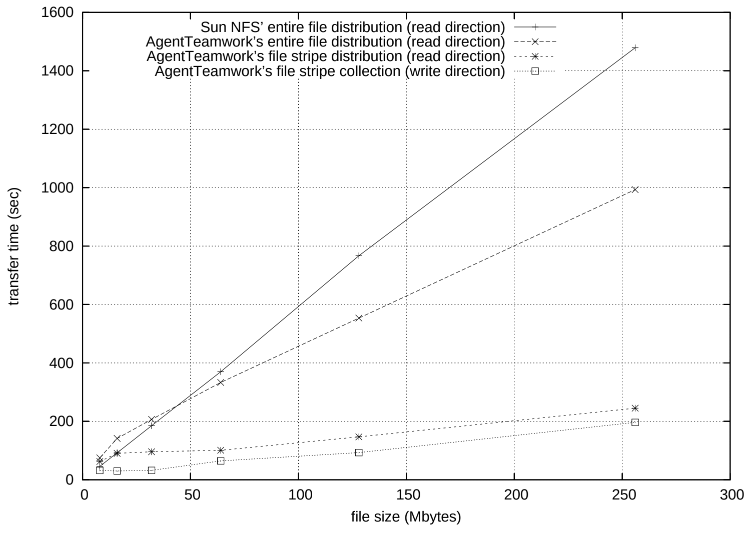


Figure 15: Performance of file transfer

where 64 file stripes are written by and sent from their respective sentinels up to the commander agent that stops the timer upon a completion.

Figure 15 compares these five test cases. AgentTeamwork's file duplication ran faster than Sun-NFS and user-level collective I/O when broadcasting a 64MB or a larger file, while performing worst below 64M bytes. This is because, for a larger file, AgentTeamwork not only mitigates disk access overheads similarly to Sun NFS' server caching and collective I/O, but also distributes file duplication overheads across agents in their hierarchy. However, for a smaller file, each disk access and even each file duplication become negligible as compared to repetitive file relays through an agent hierarchy.

AgentTeamwork's file-stripe distribution performed 6.03 and 4.05 times faster than Sun NFS' and AgentTeamwork's file duplication respectively when a random-access file is 256MB long. Obviously, the more user processes the more parallelism can be expected when distributing file stripes. Notable in Figure 15 is that file-stripe collection performed even better than distribution, (more specifically 2.99 and 1.25 times better when handling a 16MB and 256MB random-access file respectively). We explain that this distinctive performance was resulted from two file-collection paths, (described in Section 2.4.3), which successfully avoided message congestion.

3.5 Job Check-Pointing Overheads

To evaluate AgentTeamwork's check-pointing overheads, we have considered the following three test cases: (1) *non IP-caching deployment*: relays all execution snapshots from one agent to another through a hierarchy and delivers them to one bookkeeper; (2) *IP-caching deployment with 1 bookkeeper*: allows each sentinel to cache the corresponding bookkeeper's IP address, while only one bookkeeper maintains all snapshots; (3) *IP-caching deployment with 2 bookkeepers*: allows each sentinel to cache the corresponding bookkeeper's IP address, where two hosts are allocated to bookkeepers, each maintaining snapshots from *cluster-R* and *cluster-I* respectively.

As shown in Figure 16, the non IP-caching deployment shows a super-linear increase of its overhead. There are two reasons. One is that all snapshots had to be funneled through the commander agent.

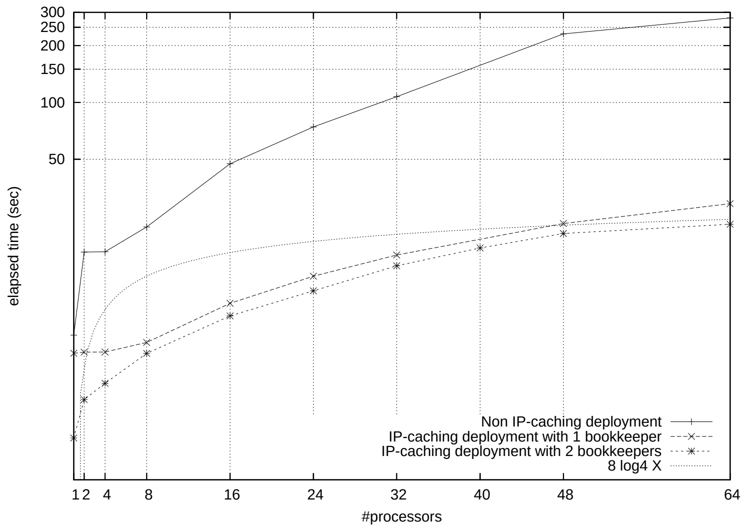


Figure 16: Job deployment effect by snapshot maintenance

Test cases	File distribution time
Test 1: no agent termination	262.60 seconds
Test 2: rank-0 termination	293.57 seconds
Test 3: rank-11 termination	266.68 seconds

Table 2: Agent recovery overheads

The other is that each gateway itself must take periodical snapshots including all *GridTcp* messages passing through it, which burdens the gateway with relaying not only its own but also the descendant gateways' snapshots. On the other hand, the IP-caching deployment with one bookkeeper has drastically improved its job deployment performance by allowing execution snapshots to be delivered directly to a bookkeeper. Furthermore, the IP-caching deployment with two bookkeepers has balanced snapshot maintenance between their hosts to demonstrate the best performance mostly bounded by $log_4 N$.

We have also assessed AgentTeamwork's file recovery overheads by distributing a 256MB file to 12 computing nodes with 8MB fragmentation and intentionally terminating a sentinel agent when it has received only the first two file messages, (i.e., the first 16MB). We considered three test cases: (1) killing no sentinel agent, (2) killing the rank-0 sentinel, and (3) killing the rank-11 sentinel. Test 1 simply distributes an entire 256MB file. Test 2 evaluates the largest overheads incurred from rank 0 to all the other sentinels. This is because rank 0 is located at the highest tree level among all sentinels. Test 3 evaluates the least overheads incurred by rank 11 at the lowest leaf, which thus affects no other agents.

Table 2 compares the file-transfer time of these three test cases. The results matched our estimation. Test 2 was approximately 31 seconds slower than test 1 whereas there was little difference between test 1 and test 3. Although this 31-second delay occupied only 11% of the entire file distribution, it will grow in linear to the number of sentinel agents, because each agent requests the commander agent to resend missing file messages. Therefore, we anticipate that, with 64 computing nodes, test 2's overheads would become $(31/11 \times 63) =$ 178 seconds.

4 Student Supervision

Through this AgentTeamwork project, the PI supervised 16 students. Their research and programming work is summarized in the table below and more detailed in the following list. The student types shown in the table indicate: (R) an undergraduate research assistant hired with the NSF grant; (E) an Ehime Univ. or UW Bothell exchange student; (U) a CSS undergraduate student working for course credits; and (G) an Ehime Univ. graduate student.

Period (mo/yr-mo/yr)	Students	Type	Research work
01/05 - 06/05	Zhiji Huang	R	implemented mpiJava with Java/GridTcp sockets.
01/05 - 06/05	Enoch Mak	R	implemented AgentTeamwork's resource agent.
06/05 - 03/06	Duncan Smith	R	implemented UWAgents with Java sockets.
06/05 - 02/06	Jun Morisaki	E	implemented AgentTeamwork's sensor agent.
06/05 - 03/08	Jumpei Miyauchi	E	implemented submitGUI and parallel file transfer
10/05 - 12/05	Etsuko Sano	U	verified and documented Zhiji Huang's mpiJava.
10/05 - 03/06	Eric Nelson	G	studied about a language preprocessor.
10/05 - 03/06	Jeremy Hall	E	worked on a function-based code cryptography.
01/06 - 09/06	Emory Horvath	R	implemented inter-cluster job deployment.
04/06 - 09/06	Cuong Ngo	R	implemented an XML-based resource database.
06/06 - 03/07	Solomon Lane	R	evaluated multi-cluster job deployment.
10/06 - 03/07	Joshua Phillips	ER	enhanced AgentTeamwork's language support.
04/07 - 08/07	Fumitaka Kawasaki	R	enhanced inter-cluster job coordination.
09/07 - 03/08	Henry Sia	R	enabled native-code execution.
09/07 - 03/08	Timothy Chuang	R	coded applications and conducted evaluation.
09/07 - 03/08	Miriam Wallace	R	wrote manuals and design sheets.

1. **UWAgents Mobile Agent Execution Platform**
   Duncan Smith, an undergraduate research assistant has totally reimplemented UWAgents using Java sockets, facilitated agent migration and communication over a cluster gateway, and secured them with SSL. He co-authored a conference paper on UWAgents with the PI [8].

2. **Language Support**
   Zhiji Huang, an undergraduate research assistant has implemented the mpiJava API in two versions: mpiJava-S and mpiJava-A. We have reported our implementation techniques and initial performance results in an IEEE PacRim'05 conference paper [4]. Thereafter, Etsuko Sano, a CSS undergraduate student has verified and documented our mpiJava implementation in order to make it available to other students in my senior courses.

   Eric Nelson, a Ehime Univ. graduate student studied about an ANTLR-based language preprocessor.

   Jeremy Hall, a CSS exchange student worked with Eric Nelson at Ehime University to research on the user program wrapper's cryptographic feature that encrypts a user program before its submission and decrypts each function for execution at run time.

   Joshua Phillips, an undergraduate research assistant enhanced GridTcp to temporarily save communication snapshots in disk for memory-saving purposes, implemented the Ateam class to allow user-initiated execution check-pointing, and shaped up the user program wrapper and mpiJava-A to handle these modifications. He also assisted Jumpei Miyauchi in implementing parallel transfer of random access files. Joshua, Jumpei, and the PI presented this work at IEEE PacRim'07 [11].

   Henry Sia, an undergraduate research assistant implemented a C++ native-code execution environment on top of the AgentTeamwork system. This environment is called from AgentTeamwork's Java-based user program wrapper, runs a C++ compiled program, and calls back the wrapper for using GridTcp and GridFile.

3. **Resource and Sensor Agents**
   Enoch Mak, an undergraduate research assistant has ported an Xindice database interface program to eXist, developed an agent-to-node mapping algorithm, and coded the resource agent so that it can pass an arbitrary number of candidate computing nodes to the commander agent. For details of our agent-to-node mapping algorithm, refer to Enoch Mak's final report and the PI's power-point file used for his colloquium talk at Keio University, both available from: *http://depts.washington.edu/dslab/AgentTeamwork/index.html*.

   Jun Morisaki, an Ehime Univ. exchange student extended Enoch's work, separated the resource-monitoring feature from the resource agent, and enhanced it in the sensor agent.

   Cuong Ngo, an undergraduate research assistant enhanced the sensor agent's deployment feature that enabled agents to migrate over computing nodes of multiple clusters and to check their resource status. He also designed an XML-based resource database manager from scratch, extended its function so as to maintain the resource information of clusters and their internal computing nodes, and implemented an applet-based GUI to the database. Furthermore, Cuong enhanced the resource agent to pass a list of available clusters as well as single desktops to a commander agent. We presented AgentTeamwork's resource management at IEEE PacRim'07 [7].

4. **GUI and Parallel File Transfer**
   Jumpei Miyauchi, an Ehime Univ. exchange student first prototyped AgentTeamwork's GUI with Java applets and implemented parallel transfers of user files through an agent hierarchy between a user and each remote process. Admitted to Ehime University's graduate school, he remotely worked on parallel transfer of and consistency maintenance on random access files for AgentTeamwork as well as revising its GUI. Jumpei and I presented a conference paper at the SDMAS'07 workshop and published a journal paper from Journal of Supercomputing, Springer [6].

5. **Inter-Cluster Job Coordination**
   Emory Horvath, an undergraduate research assistant implemented inter-cluster job deployment, check-pointing, and resumption in a hierarchy of sentinel agents as detailed in Section 3.3. He also identified the code common to all the agents and shaped it up in one utility class.

   Solomon Lane, another undergraduate research assistant installed Globus, OpenPBS, and MPI-G over two 32-node clusters, (i.e., cluster-R and cluster-I) to compare AgentTeamwork with these common middleware tools in terms of job deployment.

   Fumitaka Kawasaki, an undergraduate research assistant debugged and enhanced AgentTeamwork's inter-cluster job coordination mechanism, so that the system can mark off faulty clusters and nodes from the local XML database and retrieve different available resources from the database.

6. **Applications and Performance Evaluation**
   Solomon Lane, an undergraduate research assistant coded two scientific applications such as molecular dynamics and Schroedinger equation in mpiJava and ported them as well as a parallelized Mandelbrot program to AgentTeamwork's mpiJava-A. He evaluated the preliminary performance of AgentTeamwork's inter-cluster job coordination with these applications.

   Timothy Chuang, an undergraduate research assistant took over Solomon's work and ported molecular dynamics, Schroedinger's wave simulation, Mandelbrot, distributed grep, and matrix multiplication to both AgentTeamwork and MPI-G2. Using these applications, he measured AgentTeamwork's performance for job deployment, check-pointing, and scalability.

7. **Technical Writing**
   Miriam Wallace, an undergraduate research assistant worked as a technical writer in our project so as to prepare user manuals and design sheets for both AgentTeamwork and UWAgents.

5 Dissemination

Through this NSF-funded project, we published 2 journal papers, presented 7 conference papers, (all peer-reviewed), and gave colloquium presentations at 6 different universities in US and Japan.

5.1 Publications

1. Munehiro Fukuda, Koichi Kashiwagi, Shinya Kobayashi, "The Design Concept and Initial Implementation of AgentTeamwork Grid Computing Middleware", In Proc. of IEEE Pacific Rim Conference on Communication, Computers, and Signal Processing - PACRIM'05, pages 255–258 Victoria, BC, August 24–26, 2005
2. Munehiro Fukuda, Zhiji Huang, "The Check-Pointed and Error-Recoverable MPI Java Library of AgentTeamwork Grid Computing Middleware", In Proc. of IEEE Pacific Rim Conference on Communication, Computers, and Signal Processing - PACRIM'05, pages 259–262 Victoria, BC, August 24–26, 2005
3. Munehiro Fukuda, Duncan Smith, "UW Agents: A Mobile Agent System Optimized for Grid Computing", In Proc. of the 2006 International Conference on Grid Computing and Applications in conjunction with PDPTA06, Las Vega, NV, pages 107-113, June 26-29, 2006
4. Munehiro Fukuda, Koichi Kashiwagi, Shinya Kobayashi, "AgentTeamwork: Coordinating Grid-Computing Jobs with Mobile Agents", In Special Issue on Agent-Based Grid Computing, International Journal of Applied Intelligence, Vol.25 No.2 pages 181-198, October 2006
5. Jumpei Miyauchi, Munehiro Fukuda, Joshua Phillips, "An Implementation of Parallel File Distribution in an Agent Hierarchy", In Proc. of the 2007 International Workshop on Scalable Data Management Applications and Systems (in conjunction with PDPTA'07), Las Vegas, NV, pages 690-696, June 25-28, 2007
6. Munehiro Fukuda, Emory Horvath, Solomon Lane, "Fault-Tolerant Job Execution over Multi-Clusters Using Mobile Agents", In Proc. of the 2007 International Conference on Grid Computing and Applications, Las Vegas, NV, pages 123-129, June 25-28, 2007
7. Joshua Phillips, Munehiro Fukuda, Jumpei Miyauchi, "A Java Implementation of MPI-I/O-Oriented Random Access File Class in AgentTeamwork Grid Computing Middleware", In Proc. of IEEE Pacific Rim Conference on Communications Computers and Signal Processing - PACRIM'07, Victoria, BC, pages 149-152, August 22-24, 2007
8. Munehiro Fukuda, Cuong Ngo, Enoch Mak, Jun Morisaki, "Resource Management and Monitoring in AgentTeamwork Grid Computing Middleware", In Proc. of IEEE Pacific Rim Conference on Communications Computers and Signal Processing - PACRIM'07, Victoria, BC, pages 145-148, August 22-24, 2007
9. Munehiro Fukuda, Jumpei, "An Implementation of Parallel File Distribution in an Agent Hierarchy", In Special Issue on Scalable Data Management Applications and Systems, Journal of Supercomputing, accepted on February 25, 2008.

5.2 Colloquia

1. "AgentTeamwork: Mobile-Agent-Based Middleware for Distributed Job Coordination", Colloquium for the Novel Computing Project & Multimedia Databased Laboratories, Faculty of Environmental Information, Keio University, December 20, 2005
2. "Grid Computing Using Mobile Agents", Colloquium at IEEE Shikoku-Japan Section , Ehime University, December 22, 2005
3. "Parallel Job Deployment and Monitoring in a Hierarchy of Mobile Agents", CSS Speaker Series Colloquium, Computing and Software Systems, University of Washington, Bothell, May 25, 2006

4. "Parallel Job Deployment and Monitoring in a Hierarchy of Mobile Agents", Workshop Presentation in Messenger's Research Group, Department of ICS, University of California, Irvine University, June 23, 2006
5. "Fault Tolerant Job Execution over Multi-Clusters Using Mobile Agents", Colloquium at Department of Computer Science, University of California, June 29, 2007
6. "Parallel Job and File Distribution in an Agent Hierarchy", Workshop Presentation at Multi-Database and Multimedia Database Research Group, Faculty of Environmental and Information Studies, Keio University, December 18, 2007
7. "Parallel Job and File Distribution in an Agent Hierarchy", Colloquium at Open-Source Software System Laboratory, School of Computer Science, Tokyo University of Technology December 21, 2007

5.3 Contribution to Partner's Publication

1. Shinya Kobayashi, Shinji Morigaki, Eric Nelson, Koichi Kashiwagi, Yoshinobu Higami, Munehiro Fukuda, "Code Migration Concealment by Interleaving Dummy Segments", In Proc. of IEEE Pacific Rim Conference on Communication, Computers, and Signal Processing - PACRIM'05, pages 269–272 Victoria, BC, August 24–26, 2005

6 Budget Activities

This section reports the PI's budget activities in terms of equipment purchases, his salary for course release, student salary, and expenses for his trip

6.1 Equipments

The following table details the purchase of our Giga-Ethernet cluster of 24 DELL computing nodes. Since the equipment budget line was cut off to less than a half of our original estimation, we purchased this cluster system in support from the PI's departmental budget, (i.e., $2536.63). The deficit has been compensated with part of the indirect cost to be returned to the department.

Description	Price	Quantity	Amount
3.2GHz/1MB Cache, Xeon 800MHz Front Side Bus	$762.89	24	$18,309.36
16Amp, Power Distribution Unit120V, w/IEC to IEC Cords	$89.00	2	$179.00
RJ-45 CAT 5e Patch Cable, Snagless Molded - 7 ft	$1.61	28	$45.08
PowerConnect 2624 Unmanaged Switch, 24 Port GigE with 1 GigE/SFP Combo Port	$315.92	1	$315.92
Tax			$1,653.27
Total			$20,501.63
Alloted amount			$9,866.00
Departmental support			$2,536.63
Difference			-$8,099.00

6.2 PI's Salary

As shown in the following table, the PI used his research salary for six times of course release in total, so that his teaching responsibility was reduced to one course per each quarter during this entire awarding period.

Quarters	course releases	Salaries
Spring 05	One CSS course	$12,485.00
Autumn 05	One CSS course	$12,982.00
Winter 06	One CSS course	$12,982.50
Winter 07	One CSS course for autumn 06, but actually paid in wi07	$13,509.00
Spring 07	One CSS course	$13,509.00
Autumn 07	One CSS course	$15,117.00
Total		$80,584.50
Alloted amount		$76,418.00
Difference		-$4166.50

6.3 Student Salary

The PI has hired 12 undergraduate students for their research commitment to the AgentTeamwork project.

Name	Research Item	Hourly	Working hours	Salaries
Zhiji Huang	mpiJava	$14.00	200hrs in wi05, 120hrs in sp05	$5,264.00
Enoch Mak	Resource agent	$14.00	176hrs in wi05, 204hrs in sp05	$5,320.00
Duncan Smith	UWAgents	$14.00	131.5hrs in Su05, 228.5hrs in au05	$5,040.00
Jeremy Hall	Code cryptography	$14.00	43hrs in au05	$602.00
Duncan Smith	UWAgents	$14.00	40hrs in wi06	$560.00
Emory Horvath	Job deployment	$14.00	165, 145, & 68hrs in 3 quarters	$5292.00
Cuong Ngo	Resource database	$14.00	192hrs in sp06, 208hrs in su06	$5600.00
Solomon Lane	Performance	$14.00	120hrs in au06	$1680.00
Joshua Phillips	Language & files	$14.00	248.55hrs in au06	$3479.00
Joshua Phillips	Language & files	$14.00	199hrs in wi07	$2786.00
Solomon Lane	Performance	$14.00	111hrs in wi07, 95hrs in sp07	$2884.00
Fumi. Kawasaki	Multi-cluster job	$14.00	305hrs in sp07	$4270.00
Fumi. Kawasaki	Multi-cluster job	$15.00	101hrs in su07	$1515.00
Timothy Chuang	Application dev.	$15.00	190hrs in au07	$2850.00
Miriam Wallace	Technical writing	$15.00	32.46hrs in au07	$486.89
Henry Sia	Native code exec.	$15.00	208hrs in au0 (paid by RCR)	$0.00
Additional student Salary actually paid for years 2005 and 2006				$903.11
Total				$48,532.00
Alloted amount				$52,443.00
Difference				$3,911.00

Note that Miriam Wallace actually worked for 49.75 hours, among which 32.46 hours and 17.29 hours were paid by the direct cost and the RCR (research-cost recovery) money respectively. Henry Sia was covered by RCR, too. The total amount of the PI's and students' salary was $12,911.65 whose deficit from $12,8861.00 was $255 only.

6.4 Travels

The PI used this travel budget to give conference/research talks at PacRim 05, Keio University, and Ehime University in year 2005; at UC Irvine and GCA 06 in year 2006; and GCA 07, SDMAS 07, UC Davis, and PacRim 07 in year 2007.

Trip (dates)	Tasks	Amount
Two paper presentations at IEEE PacRim 05 (8/24/05–8/26/05)	Victoria registration fee	$331.42
	Victoria registration fee	$163.75
	Ferries to/from Victoria	$119.50
	Encumbered per diem (two nights)	$527.82
Colloquia at Keio & Ehime Universities (12/14/05-1/3/06)	Flights to/from Tokyo Japan	$771.00
	Flight between Tokyo and Ehime	$274.45
	Encumbered per diem (one night)	$257.00
A research talk at UC Irvine (6/23/06–6/24/06)	Flight from SEA to SNA	$242.30
	Car rental (one day)	$46.02
	Encumbered per diem (one night)	$106.23
A paper presentation at GCA 06 (6/25/06–6/27/06)	Registration fee	$495.00
	Flight from SNA through LAS to SEA	$236.10
	Local transportations	$15.00
	Encumbered per diem (two nights)	$212.47
A talk at GCA 07, SDMAS 07, and UC Davis (6/26/07–6/29/07)	Registration fee (for the PI)	$545.00
	Registration fee (for a student)	$395.00
	Flight from SEA to LAS and SFO	$390.10
	Car rental (two days)	$96.63
	Encumbered per diem (4 nights)	$523.32
	Other expense	$62.17
Two paper presentations at IEEE PacRim 07 (8/23/07–8/24/07)	Registration fee (for the PI)	$425.87
	Registration fee (for a student)	$473.19
	Auto Mileage (to Anacortes, WA)	$92.15
	Encumbered per diem (1 night)	$161.77
	Other expense (including ferry)	$71.95
Other cost (Freight, express, and contractual service)		$219.00
Total		$7,254.21
Alloted amount		$7,500.00
Difference		$245.79

7 Post-Award Plan

At present, AgentTeamwork is being used and enhanced by the PI and his research assistants. From this April, the PI will release the system to his research collaborators. Two promising users are:

1. **UWB Brain Grid**
   The UW Bothell Biocomputing Library led by Prof. Mike Stiber has supported the initial development and modification of a "liquid state machine" (LSM) simulator [12] that models the activity of biological neural network grown in culture. However, its use is limited to simulation of small neural networks due to its computation-intensive nature. The UWB brain grid project is intended to allow LSM users to simulate networks with 100 times as many cells. We will start this project with completing our implementation of a native-code execution environment on top of AgentTeamwork, thereafter parallelize the LSM simulator with MPICH, then run it over 64 Linux computing nodes, port AgentTeamwork to Windows, and ultimately extends uses of the LSM simulator to on-campus Windows machines.

2. **Super-Scalable Web Services**
   The Open-Source Software System Laboratory at Tokyo University of Technology launched the Gaia project to implement a world-wide super-scalable web services with their experience in distributed file-server development. Since last year, we have discussed with Prof. Tago, the Gaia

project leader about a possible use of AgentTeamwork technologies to support the Gaia system implementation as its infrastructure.

8 Final Comments

Through this three-year RUI project, we implemented the AgentTeamwork mobile-agent-based grid-computing middleware system. The system is now going to be used by the UW Bothell Biocomputing Library. From our implementation and performance evaluation, we obtained five major finding: (1) AgentTeamwork-suitable computational granularity, (i.e., 40,000 doubles $\times$ 10,000 floating-points) , (2) computationally-scalable applications for AgentTeamwork (such as master-workers and heartbeat-type programs), (3) multi-cluster job deployment in a logarithmic order, (4) parallel file distribution and collection in an agent hierarchy, and (5) the worst job resumption overheads increased in proportional to the total number of computing nodes.

As mentioned in Section 7, the PI will seek for opportunities of collaborative work that can not only use AgentTeamwork as a computational infrastructure but also continue enhancing the system for its practical use.

References

[1] Munehiro Fukuda. NSF SCI #0438193: Annual report for year 2005. Annual report, UW Bothell Distributed Systems Laboratory, Bothell, WA 98011, January 2006.

[2] Munehiro Fukuda. NSF SCI #0438193: Annual report for year 2006. Annual report, UW Bothell Distributed Systems Laboratory, Bothell , WA 98011, January 2007.

[3] Munehiro Fukuda, Emory Horvath, and Solomon Lane. Fault-tolerant job execution over multi-clusters using mobile agents. In *Proc. of the 2007 International Conference on Grid Computing and Applicaitons – CGA'07*, pages 123–129, Las Vegas, NV, June 2007. CSREA.

[4] Munehiro Fukuda and Zhiji Huang. The check-pointed and error-recoverable MPI Java library of AgentTeamwork gird computing middleware. In *Proc. IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing - PacRim'05*, pages 259–262, Victoria, BC, August 2005. IEEE.

[5] Munehiro Fukuda, Koichi Kashiwagi, and Shinya Kobayashi. AgentTeamwork: Coordinating grid-computing jobs with mobile agents. *International Journal of Applied Intelligence*, Vol.25(No.2):181–198, October 2006.

[6] Munehiro Fukuda and Jumpei Miyauchi. An implementation of parallel file distribution in an agent hierarcy. *Journal of Supercomputing*, DOI(10.10007/s1127-008-0194-0):online, March 2008.

[7] Munehiro Fukuda, Cuong Ngo, Enoch Mak, and Jun Morisaki. Resource management and monitoring in AgentTeamwork grid computing middleware. In *Proc. of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing – PacRim'07*, pages 145–148, Victoria, BC, August 2007. IEEE.

[8] Munehiro Fukuda and Duncan Smith. UWAgents: A mobile agent system optimized for grid computing. In *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*, pages 107–113, Las Vegas, NV, June 2006. CSREA.

[9] Message Passing Interface Forum. *MPI-2: Extention to the Message-Passing Interface*, chapter 9, I/O. University of Tenessee, 1997.

[10] mpiJava Home Page. http://www.hpjava.org/mpijava.html, accessible as of February 2008.

[11] Joshua Phillips, Munehiro Fukuda, and Jumpei Miyauchi. A Java Implemenation of MPI-I/O-Oriented Random Acess File Class in AgentTeamwork Grid Computing Middleware. In *Proc. of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing – PacRim'07*, pages 149–152, Victoria, BC, August 2007. IEEE.

[12] Michael Stiber, Fumitaka Kawasaki, and Dongming Xu. A model of dissociated cortical tissue. In *Proc. of Neural Coding*, pages 24–27, Motevideo, Uruguay, November 2007.

[13] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

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