NSF SCI #0438193: Annual Report for Year 2006

Mobile-Agent-Based Middleware for Distributed Job Coordination

Munehiro Fukuda

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

January 28, 2007

Abstract

This annual report presents the PI's research activities conducted in year 2006 on NSF SCI #0438193: Mobile-Agent-Based Middleware for Distributed Job Coordination. The PI has continued to investigate and to evaluate the applicability of mobile agents to grid-computing middleware by implementing and enhancing his AgentTeamwork system that facilitates job deployment, checkpointing, resumption, migration and monitoring in a hierarchy of mobile agents. In year 2006, the PI, his undergraduate research assistants, and an Ehime University graduate student focused on inter-cluster job coordination, remote resource monitoring and database construction, GUI and file transfer, language support, and application development. We have also measured the preliminary performance of AgentTeamwork's inter-cluster job coordination and parallel file transfer.

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.

Year 2006 brought in the following achievements as the mid year of our three-year NSF-granted research activities:

1. an implementation and a preliminary performance evaluation of our inter-cluster job coordination algorithm in an agent hierarchy,
2. an implementation of AgentTeamwork's dynamic job-scheduling infrastructure including runtime remote-resource monitoring and database management,
3. an implementation and a preliminary performance evaluation of AgentTeamwork's parallel and pipelined file-transfer mechanism, and
4. an enhancement of our language support as well as application development.

In addition to the research activities and preliminary results, this report will give details of the PI's student supervision, publications and budget activities in 2006 as well as his research plan for year 2007.

2 Research Activities

This section clarifies updates in AgentTeamwork's system overview and reports our implementation progress in year 2006.

2.1 System Overview

The following descriptions highlight new components and features with underlines so as to distinguish them from those implemented in year 2005.

AgentTeamwork is grid-computing middleware that uses mobile agents to deploy a user program to and monitor its execution at idle computing nodes [2]. A new node can join the system by running a UWAgents mobile-agent execution platform to exchange agents with others [3]. The system distinguishes several types of agents such as commander, resource, sensor, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, resource monitoring, job deployment and resumption, and job-execution bookkeeping respectively.

A user submits a new job with a commander agent that spawns a resource agent. To find a collection of remote machines fitted to resource requirements, the resource agent locally retrieves appropriate resource information from AgentTeamwork's XML database as well as downloads new information from a remote ftp server if necessary[1]. Thereafter, the resource agent dispatches a collection of sensor agents in a hierarchy to remote clusters and desktop computers that have not been allocated to user computations.

Given a collection of destinations, the commander agent spawns a pair of sentinel and bookkeeper agents, each hierarchically deploying as many children as the number of the destinations. Each sentinel launches a user process at a different machine with a unique MPI rank, takes a new execution snapshot periodically, sends it to the corresponding bookkeeper, monitors its parent and child agents, and resumes them upon a crash. A bookkeeper maintains and retrieves the corresponding sentinel's snapshot upon a request. User files and the standard input are distributed from the commander to all the sentinels along their agent hierarchy, while outputs are forwarded directly to the commander and displayed through the AgentTeamwork's GUI.

[1] The current implementation accesses *ftp.tripod.com* to download new XML files registered within 24 hours.

A user program is wrapped with a user program wrapper, one of the threads running within a sentinel agent. The wrapper presents a message-recording and error-recoverable TCP library named *GridTcp* to a user program, and serializes its execution snapshot in bytes including in-transit messages. Although an application can be coded with GridTcp similarly to the direct use of Java sockets, AgentTeamwork provides a user with a fault-tolerant version of the mpiJava API [5] that has been actually implemented with GridTcp.

2.2 Implementation

In the following, we explain the progress and status of our work items listed in Section 7 "Plan for Year 2006" of our last year's annual report [1] in the same order.

2.2.1 Inter-Cluster Job Coordination

AgentTeamwork used to support job coordination over only a collection of desktop computers. In other words, the system was not able to deploy a job to multiple clusters over their gateways, although the GridTcp library facilitated inter-cluster communication at a user level. The challenges in inter-cluster job coordination are three-fold: (1) to allow mobile agents to migrate over a gateway to each cluster computing node, (2) to develop an algorithm of efficiently mapping a hierarchy of sentinel agents to a collection of clusters, and (3) to minimize the overhead of over-cluster communication among agents that repeat sending a ping message and an execution snapshot to others.

The first challenge was resolved by revising UWAgents so that agents gained a new capability of traveling between separate networks or clusters and even sending messages to those on the other network. Note that the details of UWAgents' inter-cluster migration were described in [3]. 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];
gateways[0] = "gateway_1";
gateways[1] = "gateway_2";
hop("node1", gateways, "nextFunction", args );
```

The second challenge was addressed 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 1 describes an implementation of our inter-cluster job coordination. 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.

The third challenge was overcome by having an agent temporarily cache its communication counterpart's IP address after their first communication as far as they reside in the same network domain. With this address caching mechanism, an agent can send messages to its final destination or its gateway agent rather than relay them through the ascendant and/or descendant agents in the hierarchy.

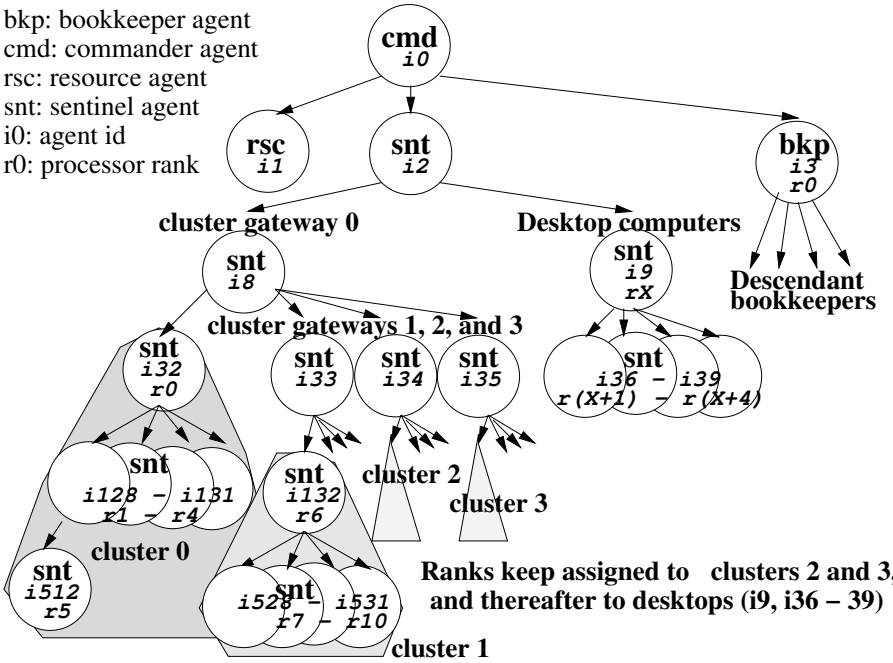

Figure 1: Inter-cluster job distribution in AgentTeamwork

2.2.2 Dynamic Job Scheduling

We have first focused on providing an infrastructure to facilitate dynamic job scheduling. It includes two components: (1) the sensor agent that migrates to a remote site and periodically collects its resource status and (2) the resource agent that maintains such runtime resource information on an XML-based database and generates a runtime collection of remote IP names fitted to a given job execution.

Sensor Agent
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.

Resource Agent
We have enhanced the resource agent to create and to access a database for runtime information on remote resources. More specifically, the resource agent downloads new XML files from a shared ftp

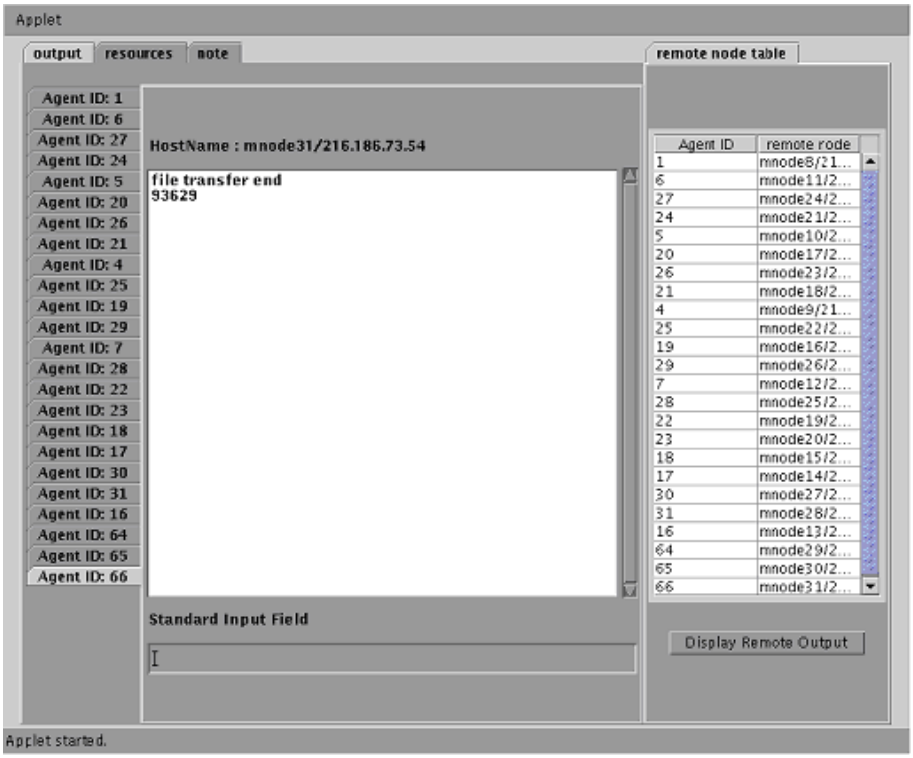

Figure 2: A GUI output

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 fitted to a given job execution.

2.2.3 GUI and File Transfer

Figure 2 shows our applet-based GUI 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.

Upon receiving a new job request, the GUI creates and executes a shell script that launches a commander agent as passing job execution parameters. Thereafter, the commander agent reads input data files and the standard input, packetizes those file streams, and distributes packets to each sentinel agent through its agent hierarchy. This file transfer avoids multiple reads of the same file on a user site, duplicates file fragments at each tree level of sentinel agents, and sends them in pipeline so that a remote user process can start its computation without waiting for an entire file.

On the other hand, output files and the standard output are forwarded from each sentinel directly to the commander agent that writes them in a user-local disk and/or prints them out in a GUI window.

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. GridFile can also maintain user files in the */tmp* disk space at each remote node if they are too large to place in memory. We are currently revising GridFile to carry these */tmp* files to a new site where a user process has migrated.

```
1   import AgentTeamwork.Ateam.*;
2   public class MyApplication extends AteamProg {
3     private int phase;
4     public MyApplication(Object o){} // system-reserved constructor
5     public MyApplication( ) {        // user-own constructor
6       phase = 0;
7     }
8     private boolean userRecovery( ) {
9       phase = ateam.getSnapshotId( );// check the snapshot version
10    }
11    private void compute( ) {        // real computation
12      ...;
13      ateam.takeSnapshot(phase);     // user-initiated check-pointing
14      ...;
15    }
16    public static void main( String[] args ) {
17      MyApplication program = null;
18      if ( ateam.isResumed( ) ) {    // application resumed in the middle
19        program = (MyApplication)
20           ateam.retrieveLocalVar( "program" );
21        program.userRecovery( );
22      } else {                       // application started from the top
23        program = new MyApplication( );
24        ateam.registerLocalVar( "program", program );
25      }
26      program.compute( );            // call real computation
27  } }
```

Figure 3: User-specific check-pointing code

As of December 2006, random access files cannot be modified among multiple processes, which is our current work item and will be implemented in the first half of year 2007.

2.2.4 Language Support

AgentTeamwork assumes that each application should be compiled to or directly coded in a collection of functions named *func_id* where *id* is a zero-based index. Launched from each sentinel agent, the user program wrapper first calls *func_0*, then repeats calling a different *func_id* indexed by the return value of its previous function, and ends in the function whose return value is -2. (See Figure 3 in our annual report for year 2005 [1].) The wrapper also takes an execution snapshot at each transition from one to another function call.

We have implemented a function-based cryptography in the user program wrapper for the purpose of securing an execution of a given user program at a remote site. The user program wrapper uses Javassist [4] to encrypt all *func_id* functions at a job submission and to decrypt each function on demand by directly manipulating byte code of a user application.

In addition to the *func_id*-based code framework, we have facilitated a more efficient coding style where users can code their applications as specifying variables and code portions to be check-pointed. Figure 3 shows such an example that starts from *main( )* as an ordinary user program (line 16), instantiates application objects (line 23), recovers necessary variables from the latest snapshot if resumed (lines 19–20), and check-points its intermediate execution where it needs to do so (line 13).

2.2.5 Applications

Besides the Java Grande MPJ benchmark programs [6], we have ported the following three programs to mpiJava [5] and thereafter to AgentTeamwork:

Programs	Descriptions	Major communication
Wave2D.java	2D Schroedinger Equation	SendRecv with neighboring ranks
MD.java	2D Molecular Dynamics	SendRecv with neighboring ranks
Mandel.java	Mandelbrot	Master-slave communication with Bcast, 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). Using those three applications, we are currently conducting performance evaluation of inter-cluster job coordination over two 32-node clusters, (thus over 64 nodes in total).

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.

2.2.6 Resource Database

We have implemented our own XML database instead of using the eXist open software for the purpose of facilitating dynamic job-scheduling in the database. Our database implementation consists of the following components:

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*.
5. **ResousrceAgent.java** has been enhanced to create initial and runtime resource collections (as described above) as well as to provide a commander agent with a list of available resources mixed with clusters and single desktops.

Our last implementation plan for the resource database is to list up available computing resource in their degree of availability and fitness to a given job request.

3 Major Findings

In year 2006, our performance evaluation has focused on job deployment and file transfer in an agent hierarchy. The former enabled job deployment over clusters, and the latter has implemented parallel and pipelined file transfer. This section shows our preliminary results.

3.1 Job Deployment in an Agent Hierarchy

We have evaluated the system performance for job deployment and termination over two cluster systems, named *medusa* and *uw-320*. As summarized in Table 1, both include 32 computing nodes, while *medusa* is faster than *uw1-320*. 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 interests in AgentTeamwork's job coordination performance are two-fold: one is effect on the entire performance by snapshot transfer to and maintenance at bookkeeper agents; and the other is effect

medusa: a 32-node cluster for research use

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

uw1-320: a 32-node cluster for instructional use

Gateway node:		
	specification	**outbound**
	1.5GHz Xeon, 256MB memory, and 40GB HD	100Mbps
Computing nodes:		
#nodes	**specification**	**inbound**
16	1.5GHz Xeon, 512MB memory, and 30GB HD	100Mbps
16	1.5GHz Xeon, 512MB memory, and 30GB HD	1Gbps

Table 1: Cluster specifications

on the performance by cluster node allocation. The following two subsections examine the system performance from these two viewpoints.

3.1.1 Effect of Snapshot Maintenance

Since AgentTeamwork deploys a job in a hierarchy of sentinel agents, its performance should be upper-bounded by $O(logN)$ in principle no matter how many computing nodes are involved in the same job. However, from our early stage of performance-tuning work, we have noticed that job execution and termination is delayed by transferring computation snapshots along an agent hierarchy to and maintaining them at fewer hosts than bookkeeper agents. This is our main motivation to conduct performance evaluation as disabling and enabling a sentinel agent's ability of caching the the corresponding bookkeeper's IP address as well as increasing the number of hosts where bookkeeper agents maintain snapshots. To be more specific, the following three test cases have been considered:

1. *Non IP-caching deployment*: transfers all execution snapshots along an agent hierarchy to and maintains them at only one host where all bookkeepers reside.
2. *IP-caching deployment with 1 bookkeeper*: allows each sentinel to cache the corresponding bookkeeper's IP address as far as the bookkeeper is reachable directly and stays at the same host. Note that all bookkeepers are still gathered on only one host.
3. *IP-caching deployment with 2 bookkeepers*: allows each sentinel to cache the corresponding bookkeeper's IP address. Furthermore, two hosts are allocated to bookkeepers, one for bookkeepers dedicated to *medusa* and the other for *uw1-320*.

Note that all of these test cases allocate both *medusa*'s and *uw1-320*'s computing nodes evenly to a job. Figure 4 compares their performance. Since the maximum number of children each agent can create has been set to four in this experiment, we added the plotted line of $k\ log_4 N$ (where $k = 8$) in order to compare these three test cases with the ideal performance.

As shown in Figure 4, the non IP-caching deployment shows a super-linear increase of its overhead. Needless to say, this performance drawback was incurred by snapshot transfer along an agent hierarchy, where all snapshots had to be funneled through the commander agent and thus this transfer overhead is proportional to the number of computing nodes. The non IP-caching deployment even performed worse when involving two cluster gateways in a job execution, (i.e., when the number of computing nodes > 32). This is because each gateway itself takes periodical snapshots including all user-level *GridTcp* messages passing through it. The more computing nodes the larger gateway snapshot growing in proportion. For this reason, each gateway is overloaded by relaying its own large snapshots as well as linearly growing snapshots from another gateway. As a result, the overhead of snapshot transfer delayed an entire program termination in a super-linear order.

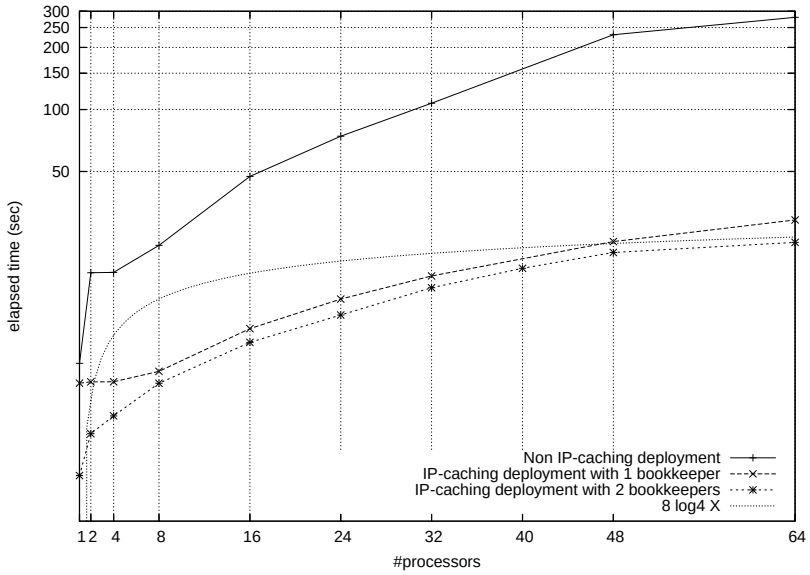

Figure 4: Job deployment effect by snapshot maintenance

Contrary to the non IP-caching deployment, 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. A special attention must be paid to snapshot transfer from a computing-node agent that runs in a private IP domain and is thus unable to reach the corresponding bookkeeper directly. Snapshots still needs to be forwarded up to the gateway agent and thereafter to be delivered directly to the bookkeeper. Despite this inevitable relay of snapshots, the IP-caching deployment has worked effectively. This proves our assumption that a gateway is overloaded by relaying large snapshots sent from another gateway rather than many but small snapshots from intra-cluster sentinels. Allocating only one host to bookkeepers, the IP-caching deployment still shows unsatisfactory performance as compared to the logarithmic curve, $log_4 N$ in particular when increasing the number of computing nodes from 48 to 64.

Finally, the IP-caching deployment with two bookkeepers has removed all the overheads discussed above to demonstrate the best performance mostly bounded by $log_4 N$.

3.1.2 Effect by Cluster Node Allocation

Our next focus is to measure the effect on the deployment performance by allocating *medusa*'s computing nodes first or *uw1-320*'s nodes first rather than allocating both nodes evenly to a job. For this evaluation, we have used the following four scenarios of job deployment and termination as increasing the number of computing nodes engaged in a job.

1. *Medusa first*: uses up *medusa*'s computing nodes first for a submitted job, and thereafter allocates *uw1-320*'s nodes to the same job if necessary.
2. *Uw1-320 first*: is the opposite allocation of *medusa first*. In other words, it uses up *uw1-320* first and thereafter *medusa*.
3. *Breath first*: allocates both *medusa*'s and *uw1-320*'s computing nodes evenly to a job.

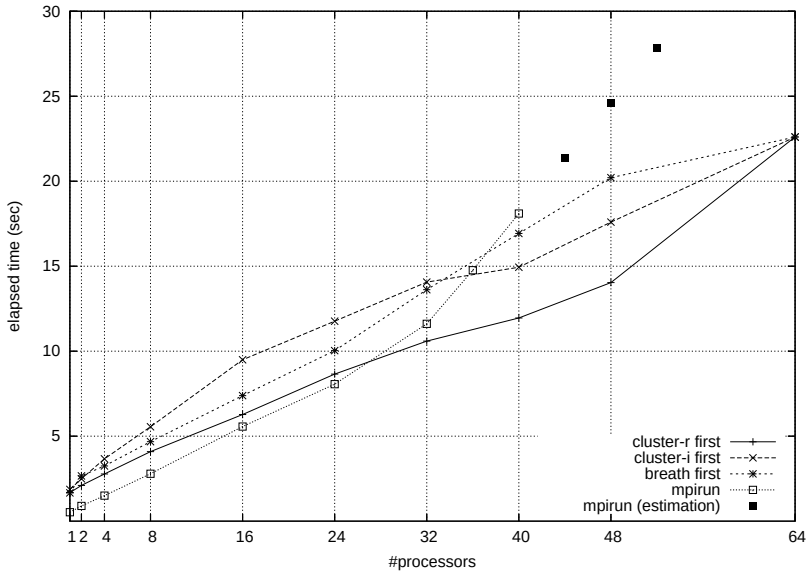

Figure 5: Job deployment effect by cluster node allocation.

4. *Mpirun*: deploys a job to *medusa* first and thereafter *uw1-320* directly using mpirun. (Note that *uw1-320*'s computing nodes are given a public IP address as well.)

Figure 5 compares their performance. Focusing on job deployment inside *medusa*, *mpirun* performed faster than AgentTeamwork's *medusa first* in most cases but indicated a linear increase of its job deployment overhead due to its repetitive *rsh* invocations. In contrast, AgentTeamwork gradually reduced a growth rate of its job deployment overhead and won against *mpirun* when using all 32 computing nodes in *medusa*.

After using up all *medusa*'s nodes, *mpirun* escalated a slope of its $O(N)$-bounded overheads. This is because each *rsh* request to *uw1-320*'s computing node had to be forwarded through the 100Mbps campus backbone and thus received additional communication overheads. To be worse, *mpirun* suffered from frequent timeouts by Kerberos *rsh* when using more than 40 computing nodes at a time. (Note that, due to this defect, Figure 5 plotted an estimated performance of *mpirun* for over 40 computing nodes.) Similarly, the *medusa first* scenario has increased its overhead growth rate toward *breath first*'s performance when using *uw1-320* as well as *medusa*. However, as discussed in Section 3.1.1, *breath first* itself alleviates its overhead growth in a logarithmic order. Therefore, we can estimate that AgentTeamwork is more node-scalable than *mpirun*.

Excluding *mpirun* from comparison, it is obvious that *medusa first* performed fastest for job deployment to the first 32 nodes; *breath first* was the next; and *uw1-320 first* yielded the slowest performance in accordance with the number of computing nodes used in the faster cluster, (i.e., *medusa*). However, once all these scenarios used both *medusa* and *uw1-320*, *breath first* admitted the slowest performance. The main reason is that *breath first* incurred more communication between two cluster systems than the other scenarios by involving the two cluster's nodes evenly in a job.

In summary, our performance evaluation has revealed the following three facts: (1) AgentTeamwork's job deployment in a logarithmic order, (2) the performance contribution of direct snapshot transfer to and distributed maintenance at bookkeeper agents, and (3) the importance of cluster node

allocation in a depth first strategy.

3.2 File Transfer in an Agent Hierarchy

We used 24 computing nodes of the *medusa* cluster to evaluate the performance of AgentTeamwork's file transfer. (See Table 1 for the specification.) The following two subsections compare AgentTeamwork with Sun NFS in their file transfer speed and evaluate the effect of AgentTeamwork's file fragmentation and pipelined transfer.

3.2.1 Comparison between AgentTeamwork and Sun NFS

We used a test case that allows each user process to access the same file whose size varies from 8M to 256M bytes. In AgentTeamwork, its file transfer time has been measured from a commander agent's injection to termination. This sequence includes (1) a commander reads a given file from its local disk; (2) the file is forwarded, duplicated, and delivered to 24 sentinels through the agent hierarchy; (3) each user process reads the file; and (4) all agents acknowledge to the commander. In Sun NFS, we have coded the corresponding mpiJava program that makes all 24 ranks read the same file and send a "complication" signal to rank 0.

Figure 6 compares AgentTeamwork and Sun NFS for their file transfer speed. AgentTeamwork performed 1.7 times faster than Sun NFS when transferring a 256M-byte file. Obviously, this large difference resulted from their file duplication schemes. AgentTeamwork accesses disk only one time and duplicates a file in each agent's memory space while relaying it through the agent hierarchy. On the other hand, Sun NFS accesses disk in response to each remote user process, which makes 24 disk accesses. However, for a smaller file with 2M through to 8M bytes, AgentTeamwork performed slower. This is because each disk access time is negligible as compared to a file relay from one to another agent in AgentTeamwork.

Figure 6 also indicates that AgentTeamwork's file-transfer speed has slowed down when increasing the data size from 128M to 256M bytes. The main reason is that each agent must relay an entire file at once, allocate more memory to maintain the file, and postpone a launch of its user application until it completely receives the file. This is our motivation to packetize a file and to transfer file data in pipeline.

3.2.2 Effect of File Fragmentation and Pipelined Transfer

Figure 7 shows the effect of file fragmentation and pipelined transfer. The legends such as 1-, 2-, and 4-splits mean that we have fragmented a given file into one, two, and four smaller packets respectively, each relayed from one to another agent through their hierarchy in a pipelined manner.

The evaluation highlighted that AgentTeamwork's file fragmentation performed 4.5 times faster than non-fragmentation and 7.7 times better than Sun NFS when sending a 256M-byte file. The results has also revealed: (1) file fragmentation does not work at all for too small files with less than 64M bytes; (2) two fragments per file perform best for 64M- and 128M-byte transfer; and (3) more fragments are necessary to transfer 256M-byte (or larger) files. Obviously, the larger file the more fragments work out to.

4 Student Supervision

Through this AgentTeamwork project, the PI supervised eight students in 2006. 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; and (G) an Ehime Univ. graduate student.

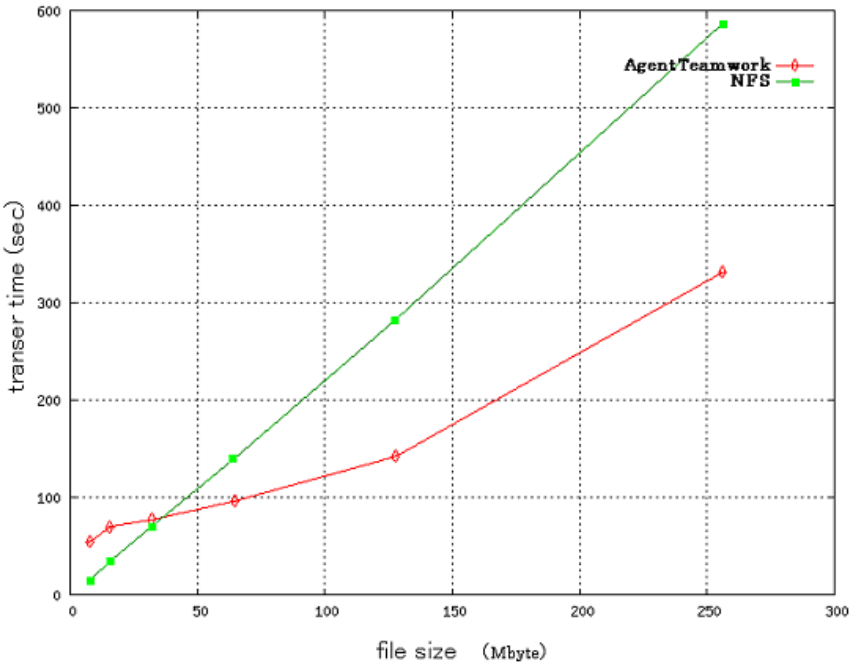

Figure 6: File transfer performance of AgentTeamwork and NFS.

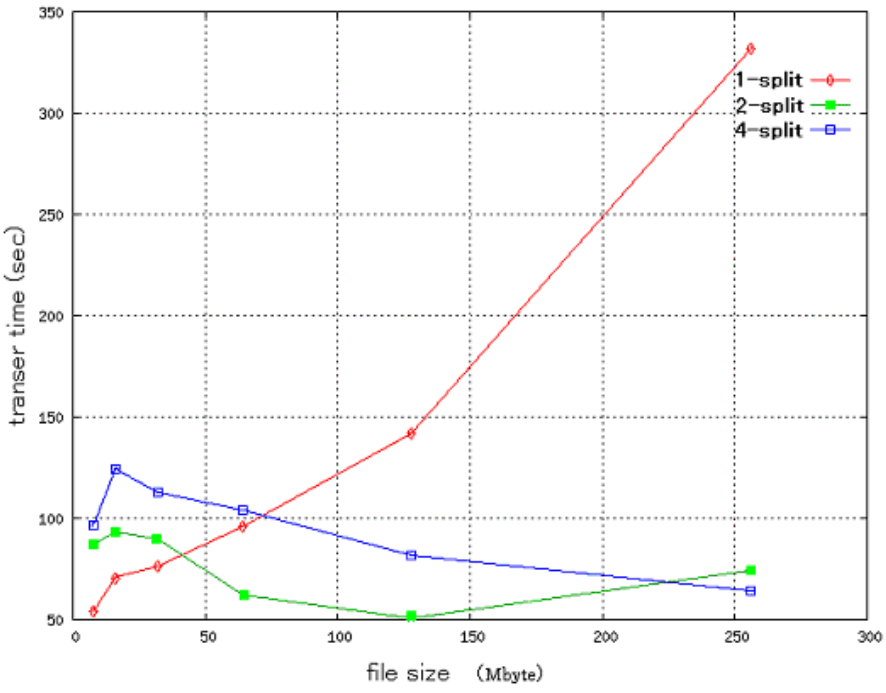

Figure 7: Effect of file fragmentation and pipelined transfer.

Period (mo/yr-mo/yr)	Students	Type	Research work
01/06 - 03/06	Duncan Smith	R	co-authored a GCA conference paper.
01/06 - 03/06	Jeremy Hall	E	worked on method-based cryptography.
01/06 - 03/06	Jun Morisaki	E	implemented AgentTeamwork's sensor agent.
01/06 - 07/03	Jumpei Miyauchi	EG	is implementing GUI and parallel file transfer.
01/06 - 09/06	Emory Horvath	R	implemented inter-cluster job coordination.
04/06 - 09/06	Cuong Ngo	R	implemented an XML-based resource database.
06/06 - 03/07	Solomon Lane	R	coded applications and is evaluating the system.
10/06 - 03/07	Joshua Phillips	ER	is enhancing AgentTeamwork's language support.

1. **UWAgents**
   Duncan Smith, an undergraduate research assistant extended his involvement in this research project till March 2006 to co-author a conference paper on UWAgents with the PI.

2. **Security**
   Jeremy Hall, a UWB exchange student worked at Ehime University on method-based cryptography of a user application. We used Javassist [4] to manipulate byte code of a given user application so as to encrypt each method at a job submission and decrypt it on demand at a remote site.

3. **Sensor Agents**
   Jun Morisaki, an Ehime Univ. exchange student implemented AgentTeamwork's resource monitoring feature in sensor agents that are hierarchically dispatched to unused remote computers so as to report their resource status through the tree up to the resource 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.

4. **GUI and File Transfer**
   Jumpei Miyauchi, an Ehime Univ. exchange student has 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 is remotely working on parallel transfer of and consistency maintenance on random access files for AgentTeamwork as well as revising its GUI.

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 2.2.1. 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., *medusa* and *uw1-320* clusters) to compare AgentTeamwork with these common middleware tools in terms of job deployment.

6. **Resource Database**
   Cuong Ngo, an undergraduate research assistant has 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. He also enhanced the resource agent to pass a list of available clusters as well as single desktops to a commander agent.

7. **Applications**
   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 is currently evaluating the performance of AgentTeamwork's inter-cluster job coordination with these applications.

8. **Language Environment**
   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 is currently assisting Jumpei Miyauchi at Ehime University in implementing parallel transfer of random access files.

5 Dissemination

We have presented one conference paper in June 2006, (and have published one journal paper in October 2006 as reported in our annual report for year 2005). The PI has also presented his project status at University of Washington, Bothell and University of California, Irvine in May and June respectively.

5.1 Publications

1. 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
2. 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

5.2 Colloquia

1. "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
2. "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

6 Budget Activities

This section reports the PI's budget activities in terms of his salary for course release, student salary, and expenses for his trips to Irvine, CA and Las Vegas, NV.

6.1 PI's Salary

As shown in the following table, the PI used his research salary for two times of course release, so that his teaching responsibility was reduced to one course per each quarter through academic year 2006. (Note that the course release in autumn 06 will be soon reflected to the UW's budget #61-1059.)

Quarters	course releases	Salaries
Winter 06	One unspecified CSS course	$12,982.50
Autumn 06	CSS430: Operating Systems (to be reflected)	$13,501.50
Total		$26,484.00
Carried-over amount from year 2005		-$987.00
Alloted amount for 2006		$25,460.00
Difference		-$2011.00

6.2 Student Salary

The PI has hired four undergraduate students for their full research commitment to and one more student for his partial involvement in the AgentTeamwork project.

Name	Research Item	Hourly	Working hours	Salaries
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
Total				$16,611.00
Carried-over amount from year 2005				$574.00
Alloted amount for year 2006				$17,472.00
Difference				$1,435.00

6.3 Travels

The PI has traveled to Irvine, CA and Las Vega, NV for a research talk at UC Irvine and a paper presentation at International Conference on Grid Computing and Applications respectively.

Trip (dates)	Tasks	Amount
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 07 (8/25/05–8/27/05)	Registration fee	$495.00
	Flight from SNA through LAS to SEA	$236.10
	Local transportations	$15.00
	Encumbered per diem (two nights)	$212.47
Total		$1,353.12
Carried-over amount from year 2005		$55.66
Alloted amount for year 2006		$2,500.00
Difference		$1,202.54

7 Plan for Year 2007

Year 2007 is the final year to make available to the public all system components and tools of the AgentTeamwork system. For this purpose, the PI is planning to hire three undergraduate research assistants, to supervise an Ehime Univ. graduate student, and to disclose the entire system to his research collaborators. For this year's paper submission, we are targeting three conferences such as GCA in conjunction with PDPTA, PacRim, and Cluster07.

1. **Random Access File and GUI**
   Since September 2006, we have been implementing a parallel version of Java 2 Ed5's RandomAccessFile.java so that AgentTeamwork's application programmers can use it based on the specification of MPI/IO's file-view features. We will complete its implementation in the first half of year 2007. In addition, we will modify AgentTeamwork's GUI so as to allow users to map a file portion to a different sentinel agent, (i.e, a different MPI rank).

2. **Dynamic Job Scheduling**
   We have so far made available AgentTeamwork's dynamic job resumption over available clusters and single desktop computers. The remaining task is to have the sentinel agent share the resource

agent's resource-sensing functions, query the resource database repetitively, and actively migrate to another site upon detecting the busy status on its current site. We are planning to complete this task in the first half of year 2007.

3. **System Validation and Refinement**
   We have independently validated and evaluated all system components except AgentTeamwork's random access file and dynamic job scheduling. As detailed above, those two components will be finished by June. Thereafter we will concentrate on the total validation and the system-wide performance evaluation in the last half year. If our time allows, we would like to implement AgentTeamwork's interface to C/C++ applications, which may be of our collaborators' interests.

4. **Dissemination**
   We will get prepared for and revise a user manual and a design note for each system component including UWAgents, GridTcp, mpiJava-A, AgentTeamwork's resource database, and all AgentTeamwork's mobile agents. We will make the system available to our collaborators at Ehime University as well as researches at UC Irvine, Tokyo University of Technology, and Technical University of Lisbon with whom the PI has interacted, as seeking for any possibility of applying AgentTeamwork's technology and knowledge to their applications.

8 Final Comments

Our project work for year 2006 has pursued the system development and implementation as planned in the previous year's annual report [1]. As described in Section 3, our work obtained two major findings, each in inter-cluster job coordination and parallel file transfers respectively in an agent hierarchy, which we will soon summarize in conference papers. While we still need to implement and modify a few system features such as random access files and dynamic job scheduling, we expect that we will complete them in the first half of year 2007 and devote ourselves to the knowledge and system dissemination in the last half year.

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, 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.

[3] 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.

[4] Javassist Home Page. http://www.csg.is.titech.ac.jp/∼chiba/javassist/.

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

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