RUI: Mobile-Agent-Based Middleware for Distributed Job Coordination

Munehiro Fukuda

May 14, 2004

1 Introduction

In spite of targeting broader user domains, grid computing still benefits only those who are privileged by gigantic institutions/companies and are free to forage for their organizational supercomputers or large-scaled cluster systems. Suited for these users, most commercial middleware systems so far have based their architecture on the master-worker model that prepares a central server to manage all computing resources, to accept user jobs, and to dispatch them to best-fitted computing nodes [10]. This is because their architecture could achieve the best performance, provided only a handful number of users invoke a bag of tasks, each embarrassingly parallel, which utilizes as many computing nodes as possible.

There is, on the other hand, another type of users who wish to remotely and mutually offer their own computing resources for their time-critical needs. For instance, researchers/professors in the same joint project may wish to conduct simulations by sharing their own workstations or cluster machines that are distributed over different campuses. Students in the same course may compete for least-loaded institutional computers from their homes. Remote desktop computer owners may wish to run multi-user game on their computers. They could generally agree to form a closed computational community, using a public Internet group, FTP, or HTTP server that however disallows them to run any user-level processes and thus does not work as a central job-administrative server. Furthermore, as seen in the above examples, their applications are not always fitted to the master-worker model. It is our ultimate goal to provide such computer owners' groups with a decentralized middleware environment that matches their computational needs.

For the last decade, mobile agents have been highlighted as a prospective infrastructure of decentralized systems used for information retrieval, e-commerce, and network management. In fact, several systems have been proposed to use mobile agents as their grid-computing infrastructure [11, 22, 2], claiming the following merits of mobile agents: their navigational autonomy contributes to resource search; their state-capturing eases user job migration; their migration reduces communication overhead incurred between their client and server machines; and their inherent parallelism utilizes idle computers. Regardless of these merits, mobile-agent-based systems have not yet outperformed other commercial grid-computing middleware systems. There are several conceivable reasons: they aim at the same user domain as commercial systems; they are based on the master-worker computation where communication takes place only at the beginning and the end of each spawned task; they funnel all mobile agents through a central administrative server that thus neutralizes their inherent parallelism; and they carry user jobs, based on weak migration that resumes the jobs from a different function rather than where they have been suspended. Therefore, it appears that mobile agents could not give more than just an alternative approach to grid middleware implementation unless they switch their target to new user domains and computational models different from other commercial systems.

Based on the above background, we have focused on a community of ordinary computer owners as our user domain, and are implementing the AgentTeamwork middleware system that applies mobile agents to the decentralized coordination of user jobs not necessarily fitted to the master-worker model. When a community user submits a job, AgentTeamwork deploys a *commander* agent that spawns as many *sentinel* agents as user processes engaged in the job. Each sentinel migrates to a different computing node where it launches and monitors a user process. The commander also spawns several *bookkeeper* agents as requested by the user. Each bookkeeper maintains a process snapshot received from sentinels, and retrieves the latest copy if other agents are accidentally killed. All agents autonomously migrate to another node if the current node no longer satisfies their resource requirements. Facilitated between a sentinel and a user process is a user program wrapper that periodically takes a process snapshot through its check-pointing functions. These functions also take a consistent cut of messages in transit between any two processes. For this purpose, a user program is preprocessed with JavaCC [13] or ANTLR [1] to automatically include check-pointing functions before its execution. Our preprocessing technique uses strong migration, guaranteeing to resume a user process where it has been suspended.

Since January 2003, the PI has carried out this project with undergraduate students of University of Washington, Bothell and his collaborators of Ehime University, Japan. We have implemented a Java-based mobile agent execution platform, the basic behaviors of agents running on top of the platform, and a user program wrapper, while we still need to develop a language preprocessor. To study its feasibility, we have also measured the preliminary performance using three Java Grande MPJ benchmark programs [21]. This proposal gives the overview, the technical merits, and the competitive performance of AgentTeamwork as well as the next three-year plan to carry out our RUI-type project.

2 Issues and Related Work

We assume that a computational community, agreed on by remote desktop/cluster owners, has the following characteristics: the community can share a central resource database in its Internet discussion group, FTP, or HTTP server, while no user processes are allowed in that server; a community user is required to download an agent execution platform and all agent programs when joining to the community; any owner may power off his/her computer or kill processes dispatched from the other owners without any advance notice; some computers such as cluster nodes may reside behind a gateway; applications will be coded in C/C++ or Java; and a user job may spawn multiple processes, each communicating with any other processes, (i.e., not limited to the master-worker model.) Given these assumptions, we discuss issues and existing solutions to have mobile agents capture process execution, maintain execution snapshots, schedule jobs, and enforce system security.

2.1 Snapshot of Communicating Processes

Various snapshot algorithms have been developed in process migration. Condor is the most well-known system to dump a process memory image to a file, (e.g., a core-dump file in Unix) and to retrieve the process from that file at its new location [6]. The major problem is inter-process communication. Upon a process migration, some software or user assistance is necessary to reestablish previous TCP connections without losing in-transit messages. The Condor MW project has focused on the master-worker model where the MW library in a master process keeps track of and retrieves connections to all its worker processes [5]. However, inter-worker communication is not retrievable due to the lack of snapshots taken at each worker.

Rock/Rack has facilitated reliable TCP connections by wrapping actual socket connections and buffering in-transit data in a user address space [24]. Using Rock/Rack, a user process can reestablish a new TCP connection to its "migrating" peer process, provided it is called back from its peer and receives that peer's IP address. Although any communication among migrating processes can be technically reconnected by combining Condor and Rock/Rack, users are forced to develop mobile-aware applications, (which periodically check occurrences of migration and call back a new IP address to a peer process.) In addition, this solution does not work if a peer process migrates over gateways.

Another snapshot problem lies in Java. If applications are coded in Java, it is impossible to capture their program counter from the Java virtual machine without modifying it. This is why most Java-based mobile agents follow weak migration in that they have to resume their execution from the top of a given function rather than where they have been suspended. The CIA project enabled strong migration in Java-based mobile agents with its language preprocessor [12]. Its technique inserts an *if* and an *else* clause in every function that includes a migration statement. During its initial execution, a mobile agent executes the *if* clause that takes a snapshot of local variables in the current function. Upon a migration, the agent chooses the *else* clause to restore the variables. This technique could be applied for resuming ordinary Java applications from where their execution has been captured, however it needs external support to save a stack of all function variables.

In AgentTeamwork, we use a language preprocessor for process check-pointing. Our preprocessing technique is an extension of the UCI Messengers' compiler that divides a user program into two different functions before and after migration, (i.e., process check-pointing) [23]. Thus, a user program is converted into a series of functions, between any two of which a user program wrapper captures the execution state. Recursions are possible by accumulating each function's local variables to the snapshot, thus emulating a function stack. Similar to Rock/Rack, in-transit messages in TCP connections are logged by a user program wrapper. The difference is that our mobile agents take care of automatic TCP re-connections over gateways, so that applications are completely relieved from mobile awareness.

2.2 Backup

The next issue is where to back up execution snapshots in preparation for future process retrieval. Catalina assigns to an application an application-delegated manager (ADM) that deploys *task* agents, each managing a process execution at a different machine [11]. ADM works as a central backup manager of all process snapshots, which would thus suffer from its performance and scalability problem. In J-SEAL2, a client user contacts an *operator*, a job-coordinating server that dispatches his/her job with a *deployment* agent to available *donator* sites where a *mediator* process launches, monitors, suspends, and resumes a user process [2]. This indicates that process snapshots are taken and maintained locally. While snapshots are distributed over different *donator* sites, it is still zero fault-tolerant to each process.

The availability of process snapshots can be generally enhanced using some replica-management algorithms: primary-copy, active-copy, and quorum-based protocols [18]. The primary-copy algorithm centralizes all snapshots into a primary server that thereafter distributes theses copies to secondary servers, which obviously suffers from performance bottleneck at the primary server. The active-copy and quorum-based protocols allow snapshots to be directly distributed to available backup servers, thus addressing performance bottleneck. However, when it comes to our AgentTeamwork system, each sentinel agent needs to keep track of all bookkeeper agents that may repeat their migration. In addition, any bookkeepers may fail in receiving process snapshots during their migration or suspension, because of which sentinels are responsible to locate their latest process snapshot.

As used in LFS [17], we could also chronologically log the difference between the latest and its previous snapshot at each bookkeeper agent. In order to prevent each bookkeeper from exhausting its local disk space, this scheme, however, requires periodical communication among all bookkeepers to identify garbage-collective old logs. While various communication strategies have been studied for this purpose, we need to tune up the communication frequency and the order of messages exchanged among bookkeepers every time they migrate to another node and thus change their communication topology.

Our snapshot maintenance in AgentTeamwork decides one-to-one mapping from a sentinel to a bookkeeper agent. Using round robin, a bookkeeper forwards to another bookkeeper every new snapshot received from the corresponding sentinel. This scheme avoids performance bottleneck as in Catalina and the primary-copy protocol, snapshot broadcast as in the active-copy protocol, and garbage-collection as in LFS. The drawback is that the latest snapshot may not be retrieved if a crashed bookkeeper has maintained it. However, given N bookkeepers, at least the $(N-1)$th previous snapshot is guaranteed for a retrieval. We anticipate that crashes are not repeated so frequently during a job execution, (although the occurrences depend on each job size), and therefore such a penalty would be still acceptable.

2.3 Job Scheduling

Centralized job scheduling collects all distributed resource information at a central scheduler that allocates appropriate resources to each job. On the other hand, decentralized job scheduling has no focal point in resource scheduling and can be, for instance, implemented by a local and a meta-scheduler at each computing node as in [19]. Jobs are locally submitted to a meta-scheduler that exchanges load information with other nodes, are dispatched to a remote node considered of as being light-loaded, and are scheduled by its local scheduler. The main drawback of decentralized scheduling is heavy communication among these meta-schedulers, because of which its performance may become even worse than centralized scheduling. One improvement is to communicate with only K remote mete-schedulers rather than all. In any cases, each computing node must repeat initiating its active contact with other nodes, which may not be accepted to all computer owners in our target community.

Condor uses a hybrid of centralized and decentralized scheduling. A central manager maintains a collection of computers as its Condor pool. The manager periodically receives a new recourse advertisement from each of these computers and returns to a user the advertisements matching his/her resource request. If there are no appropriate advertisements, the manager forwards the user's request to another Condor pool using its gateway flocking mechanism. Or, if a user belongs to multiple pools, he/she can query each manager of these pools through the Condor's direct flocking. While this scheme accurately performs gang-scheduling of idle nodes, it still needs a central manager which could not be applied to our user groups whose central server is only available for passively maintaining resource information.

In AgentTeamwork, we will use a priority-based decentralized job scheduling that does not require periodical inter-node communication or a central manager. A commander agent prioritizes its user job according to the amount of resources requested by the user. The less resources the higher priority a job receives, (i.e., the more resources the lower priority.) With this job priority, a sentinel agent migrates to a computing node listed in its itinerary, where it preempts the current job and starts its own job as far as it has a higher priority than the current job. If the sentinel's job has a lower priority, it decides to migrate to another node or wait until the priority becomes higher than all the other jobs. A job priority is incremented every migration, while it is decremented every execution check-pointing. This way allows each job to eventually acquire resources it has requested.

Our scheduling algorithm could be compared with a special case of the computational-economy model where providers offer priced computational services and users choose services affordable within their budget [3]. The difference is that we delay a large service long instead of pricing it high, so that any users in a computational community will eventually acquire their requested resources.

2.4 System Security

The following five problems must be addressed when implementing AgentTeamwork:

1. **Authorizing a user to a computational community:** We are assuming that a computational community has a central resource database in its Internet group, FTP, or HTTP server to which a user's commander agent must authorize itself at least once when using the database.
2. **Authorizing agents to a computing node and vise versa:** We use private/public key cryptography. All agents are encrypted with a community moderator's private key and decrypted with the corresponding public key in prior to their execution at a new node. On the other hand, each agent includes the agent execution platform's checksum with which it authorizes the platform to itself.
3. **Securing inter-process communication:** This can be supported by *ssh* and *SSL*.
4. **Protecting an agent from other agents:** We disallow communication among agents injected by a different user. However, an agent injected from the *shell* can recursively spawn its descendants, all allowed to communicate with each other within their same hierarchy termed an *agent domain*. Using this idea, a commander, sentinels, and bookkeepers engaged in the same user job will form an agent domain for their interaction, but do not interfere against those working on a different job.
5. **Protecting computing node from a user process and vise versa:** Most grid-computing systems wrap a user process, monitor all its system calls, and reject high-risk resource requests, which is also applicable to AgentTeamwork. Of importance is how to secure a user program from an eavesdropped node in particular during its execution. Since our language preprocessor divides a user program into check-pointed functions, we could

deceive eavesdroppers by shuffling those functions or even inserting dummy functions as maintaining the data flow. We term this idea *execution cryptography*.

In summary, we will implement two security algorithms such as domain-based agent interaction and execution cryptography, while applying existing techniques to all the other security concerns.

3 System Model

Figure 1 shows the architectural overview of AgentTeamwork. The system targets a computational community that is agreed on by remote desktop/cluster computer owners, is formed with an Internet discussion group, FTP, or HTTP server, and is moderated by a community leader. Each user uploads his/her XML-formatted account and computing resource information to the server, and downloads the AgentTeamwork software kit including a mobile-agent execution platform, all encrypted agents, and a Xindice XML database manager [20]. The user runs at least this agent execution platform and Xindice in background. The execution platform needs to download updated account/resource information from the community server upon its initialization or as its daily routine work. Of all AgentTeamwork features, this is only the one that inevitably follows a centralized scheme.

Prior to a job invocation, the program must be preprocessed and enabled for process check-pointing and migration. A new commander agent is allocated to every new job and starts on the local execution platform. It spawns a resource agent that accesses its local Xindice database to plan an itinerary of computing nodes satisfying the job resource requirements. Receiving a node itinerary, the commander deploys the same number of sentinel agents as that of user processes engaged in the job. It also deploys as many bookkeeper agents as required by the job.

Each sentinel migrates to a different computing node where it launches a user program wrapper that starts a user process, monitors it, and collects its results. The sentinel periodically sends the latest process snapshot to its corresponding bookkeeper agent. At a different node, each bookkeeper waits for and forwards a new snapshot to another bookkeeper, using round robin. Each agent monitors the local resource conditions and migrates to another computing node listed in its itinerary, once the current node no longer satisfies its resource requirements.

The commander agent takes care of resuming any sentinel/bookkeeper agent that has been terminated during its computation. To maximize the decentralized feature of AgentTeamwork, the commander monitors only one of its spawned bookkeepers. All the other agents are monitored in a distributed fashion where any pair of sentinel and bookkeeper confirms each other's existence by repeatedly receiving the latest snapshot and its corresponding acknowledgment. If a crashed agent is a sentinel, the commander resumes it from the latest log retrieved through bookkeepers. If it is a bookkeeper, it is simply re-started with an empty log. If a sentinel is migrated or resumed at a new node different from where it used to work, it reestablishes all TCP connections to and from its user process by reinitializing its user program wrapper.

The commander agent itself may be crashed, which is then detected by the bookkeeper communicating with the commander. To be worse, this bookkeeper may be crashed, which is however discovered by any other bookkeepers that eventually forward snapshots to this crashed bookkeeper, using round robin. Therefore, any agent crashes are detectable.

Finally, standard-output results are all returned as a string array to the commander that then prints them out to its client's computer display, which is our current implementation. Since the commander may migrate away from the client computer, we need to provide sentinels with other output options: sending results to the client by email, printing out results directly to the client display if possible, and exchanging data files with the client.

4 Implementation Techniques

Based on our survey on technical issues, we are currently implementing AgentTeamwork. Its software tool kit consists of (1) the underlying mobile-agent execution platform, named UWAgents, (2) commander/resource/sentinel/bookkeeper

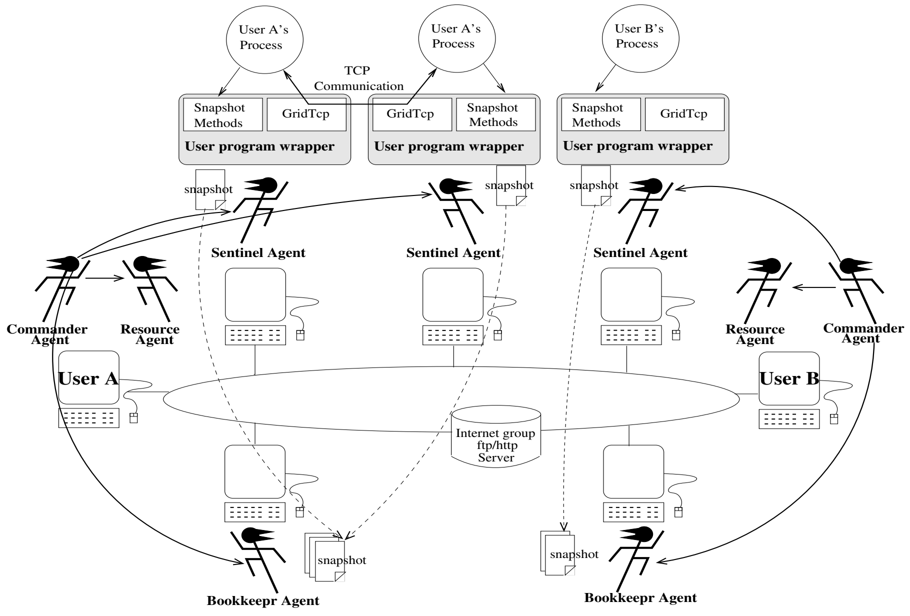


Figure 1: The Overview of AgentTeamwork.

agents running on top of UWAgents, (3) a user program wrapper, (4) a mobile-unaware TCP communication library named GridTcp, and (5) the Xindice database manager made available from the Apache XML project. In the following, we will explain each of these components.

4.1 UWAgents Execution Platform

UWAgents is a Java-based mobile-agent execution platform developed for AgentTeamwork. Each computing node executes UWPlace.class that exchanges mobile agents with other nodes. An agent is extended from UWAgent.class, is instantiated as an independent Java thread, starts with init( ), and migrates with hop( ). The hop( ) method is based on weak migration and implemented with RMI. It serializes the calling agent into a byte array, sends the array with the agent class files to the destination, and resumes the agent from a given function. The major restrictions include: (1) local I/O is not forwarded to migrating agents; (2) agents are responsible to restart Java threads by calling start( ) if they have instantiated these threads internally; and (3) they can spawn no more than 1000 child agents.

An agent submitted from the *shell*, (using the UWInject command,) forms a new agent domain identified with a pair of an IP address and a time stamp that describes where and when it has been formed. This agent is treated as a domain root, given an agent ID 0, and capable of spawning up to 999 children. All agents spawned from the same root belong to the same agent domain, receive a domain-unique positive integer as their agent ID, and are allowed to spawn 1000 children, each identified with the parent agent ID $\times$ 1000 $+$ a different sequential number from 0 through to 999. For instance, agent #1 can spawn agents with ID 1000 through to 1999. This identification scheme needs no global name server. Furthermore, two communicating agents in the same domain can locate each other by traversing their agent ID tree. For this purpose, each agent keeps track of where its parent and children have migrated. (In other words, an agent must inform its parent and children of its new position upon migration.)

The talk( ) method implements such inter-agent communication. It allows a string and/or a hash table to be sent from one to another agent. A new message must be fowarded along an agent ID tree to its destination. For better performance, the destination IP address is returned to and cached in the source agent. The consecutive messages are therefore sent directly to the destination agent as far as it stays at the same location. For instance, if agent #1 needs to communicate with agent #2001, its message is forwarded first to agent #0, then to agent #2, and finally to agent #2001. If it needs to communicate with agent #2001002, its message is directly passed to agent #2001 and thereafter delivered to agent #2001002. However, this implementation has a problem where children may not communicate with their siblings or grandparent if their parent has finished its execution, which could be solved by adapting these children to the grandparent before allowing the parent to be terminated.

We have implemented UWAgents' basic migration and communication features described above. The next step in our UWAgent implementation includes the prioritized agent execution and security features we have considered in section 2.

4.2 User Program Wrapper

AgentTeamwork must preprocess C++ and Java applications to let them include check-pointing functions that periodically take their execution snapshot. We will use the following preprocessing techniques in our ANTLR/JavaCC-based compiler design.

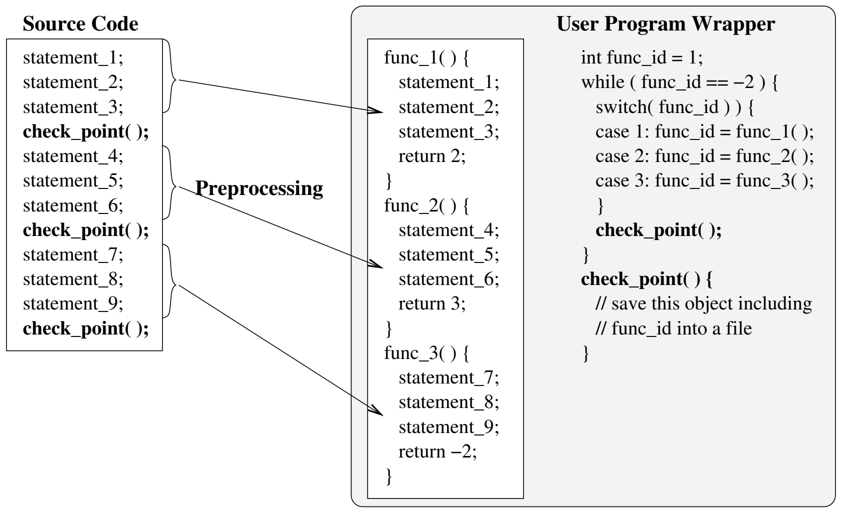
For C++ applications, we convert each of them into a mobile thread, termed an M++ thread, which we have developed in our M++ multi-agent system [9]. This thread is then inserted hop( ) functions, each migrating the calling thread to a remote computing node specified as its argument. Without any arguments, a hop( ) function simply captures and resumes the calling thread's execution state such as its CPU register contents and class data members, which is internally implemented with Unix setjmp and longjmp functions. All function calls inside an M++ thread are inline-expanded at compilation time, so that the thread does not have to carry its growing stack with it. However, our preprocessing technique has two limitations: no recursive calls and no dynamic object instantiations in a thread.

For Java applications, we convert each of them in a hierarchy of objects, all serializable from the very first object instantiated by a user program wrapper. Using this Java object serialization, all data members defined in each application are serialized into and de-serialized from a stream-oriented snapshot. The problem is that Java object serialization does not capture a program counter and a stack. To address this problem, we extend the UCI Messengers' compiler technique that converts a user program into a series of functions, between any two of which a user program wrapper captures not only its data members but also the index of the next function to execute. Figure 2 shows the simplest check-pointing example where nine sequential statements are grouped into three functions, named func_1, func_2, and func_3, each returning the index of the next function to call. The user program wrapper saves this index value as well as all the user data members, and then calls the indexed function. Upon a process crash, the wrapper retrieves the last index from the corresponding snapshot and resumes the process from the indexed function. The preprocessing technique for more complicated programs is described in [23].

4.3 Grid TCP Class

Upon its initialization, a user program wrapper instantiates an object from the GridTcp class that provides a user program with ordinary Java socket classes, as keeping track of all TCP connections established within that program. Similar to the *ssh*'s port tunneling and forwarding, GridTcp funnels all TCP connections in one connection and relays them to a third computing node.

The difference from *ssh* is GridTcp's message routing capability. AgentTeamwork dispatches a sentinel agent to and launches a user program wrapper at all gateway nodes between any two communicating user processes, where GridTcp repeatedly forwards all TCP messages to their next gateway or their final destination. For this purpose, all computing nodes involved in the same job receive a different processor rank and associate their IP address with this rank. A sentinel agent passes its user program wrapper all pairs of a rank and the corresponding IP address. With this information, the user program wrapper constructs a routing table to relay incoming messages to its neighbors. Resumed at a new node, the wrapper must receive updates in rank/IP address pairs from its sentinel agent and reflect them to its routing table entries, so that previous connections are reestablished to and from this new node.



```
statement_1;
statement_2;
statement_3;
check_point( );
statement_4;
statement_5;
statement_6;
check_point( );
statement_7;
statement_8;
statement_9;
check_point( );
```

```
func_1( ) {
  statement_1;
  statement_2;
  statement_3;
  return 2;
}
func_2( ) {
  statement_4;
  statement_5;
  statement_6;
  return 3;
}
func_3( ) {
  statement_7;
  statement_8;
  statement_9;
  return −2;
}
```

```
int func_id = 1;
while ( func_id == −2 ) {
  switch( func_id ) ) {
  case 1: func_id = func_1( );
  case 2: func_id = func_2( );
  case 3: func_id = func_3( );
  }
  check_point( );
}
check_point( ) {
  // save this object including
  // func_id into a file
}
```

Figure 2: Source program preprocessing.

A user process may lose in-transit messages during its migration or suspension. To prevent this message drop-off, GridTcp saves each connection's incoming and outgoing messages in its internal queues. The queues are captured in a process snapshot, and thus retrieved every process migration or resumption. GridTcp garbage-collects its history of these in-transit messages by exchanging a commitment message with its neighboring nodes. Our current implementation sends a commitment whenever a user program wrapper takes a new snapshot.

Figure 3 gives an example where two processes are involved in the same job, running at n1.uwb.edu and n2.uwb.edu, and identified with rank 1 and 2 respectively. When a user process migrates from n2.uwb.edu to n3.uwb.edu, their TCP connection is broken, which thus asserts an exception to their sentinel agents. The sentinel with rank 1 receives a new IP address from the one with rank 2, (i.e., n3.uwb.edu.) Thereafter, these two sentinels pass the rank2/n3.uwb.edu pair to their user program wrapper that restores in-transit messages from the latest snapshot file, updates its routing table, and reestablishes the previous TCP connection.

In summary, a user program can run multiple processes that communicate among one another using their rank, while it has no need to be aware of migration or to be based on the master-worker model.

4.4 Agent Behavioral Design

A commander agent represents each user job. In our current implementation, a commander agent and a new job must be submitted together through the UWInject command typed from the *shell* prompt. UWInject receives a job file name, arguments passed to it, and resource requirements for this job execution: the number of CPUs, CPU architecture, memory size and hard disk capacity.

Upon a job submission, the commander spawns a resource agent and sends the resource requirements to it in the form of a hash table. The resource agent starts a Xindice XML database manager if it has not yet been started, converts the hash table into a series of queries, passes them to Xindice, and returns an itinerary to the commander. The itinerary is a series of destination groups, each including the same number of computing nodes required by a job. For a parallel application, the top groups include cluster nodes so as to reduce communication overhead, while the lower groups consist of other individual desktop computers. All nodes in the same destination group are distinguished with a processor rank.

Given an itinerary, the commander agent deploys sentinel agents. Each sentinel receives this itinerary as well as a different processor rank. It traverses computing nodes with the same rank enumerated from top to down in its

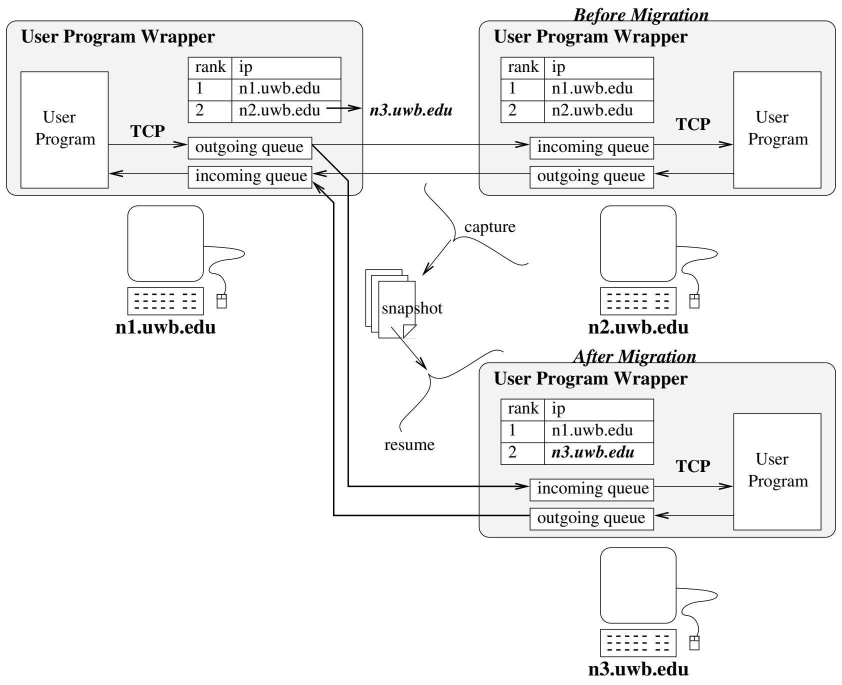


Figure 3: TCP maintenance upon migration.

itinerary until it finds the first node that can actually satisfy the resource requirements. The sentinel then launches a user program wrapper and keeps calling its receiveSnashot( ) function for the purpose of receiving from the wrapper and sending to its corresponding bookkeeper a hash table that maintains a pair of a version number and a process snapshot.

The commander also spawns bookkeeper agents, each maintaining execution snapshots received from a different sentinel, as described in section 3. The bookkeepers use the same itinerary as sentinels do for their migration, however they will not select the same destination as their corresponding sentinels. This is to prevent a pair of sentinel and bookkeeper from being killed simultaneously.

When detecting a sentinel crashed, the commander sends a snapshot retrieval query to the corresponding bookkeeper. If this bookkeeper has forwarded the latest snapshot to another agent using round robin, the retrieval query must be rerouted to the bookkeeper that actually maintains the latest version. In any cases, the commander can retrieve the latest snapshot in at most two repetitions of query forwarding.

For scalability, sentinels and bookkeepers could be spawned in a hierarchy. We actually applied this hierarchical generation to bookkeepers. However, the problem is how to resume crashed agents that used to reside in a middle of this agent hierarchy. We need to enhance the UWAgent platform so that resumed agents can automatically inform their parent and children of its new IP address.

5 Preliminary Performance

As of May 2004, we have implemented the basic features of the UWAgent execution platform, GridTcp, a user program wrapper, and mobile agents such as the commander, the resource, the sentinel and the bookkeeper. However, we have not yet implemented a JavaCC/ANTLR-based preprocessor, because of which applications must be manually converted into the version executable with the user program wrapper, (termed the AgentTeamwork version.) To evaluate AgentTeamwork's preliminary performance, we have converted the following three Java Grande MPJ benchmark programs into the ones using ordinary Java sockets (termed the Java version) and further translated them into the AgentTeamwork version: (1) PingPong: point-to-point communication, (2) Crypt: IDEA encryption, and (3): MolDyn: molecular dynamics simulation. For our performance evaluation, we used a Myrinet-2000 cluster of eight DELL workstations, each with 2.8GHz Pentium-4 Xeon processor, 512MB memory, and 60GB SCSI hard disk.

5.1 Process Communication

PingPong measures point-to-point communication performance by sending a message back and forth between two different computing nodes as increasing the message size from 8K to 512K bytes. We ran both Java and AgentTeamwork versions on two different cluster nodes to evaluate the actual communication bandwidth. For the AgentTeamwork version, we added two more different test conditions: (1) inserting another cluster node as a gateway that relays all messages exchanged between two nodes, and (2) inserting two cluster nodes as gateways where all messages must be relayed twice before delivered to the final destination.

As shown in Figure 4, the Java version's network bandwidth approached to the ideal performance as we increased the message size. On the other hand, the corresponding AgentTeamwork version showed its peak performance in its 64K-byte message transfer but marked only $20 \sim 23$% of the Java version in 256K-byte or larger message transfers. GridTcp not only funnels all Java socket connections into one but also uses one additional Java thread that reads incoming TCP messages into its buffer whenever a user program's main thread is blocked to write large messages. Frequent context switches between the user and GridTcp threads incur such communication overhead especially for large message transfers.

In terms of the message-relaying performance, one gateway insertion gave only a small impact to the bandwidth, while it displayed slower performance than the version without any gateways in small message transfers. The main reason is that large messages are pipelined and thus continuously delivered to the destination, which hides the gateway overhead and highlights only the thread context switch overhead we have discussed above. Two gateway insertions gave a larger performance drawback that however stayed in 20% slow-down of the bandwidth using one gateway.

Since our evaluation was conducted over 2Gbps Myrinet, we feel that the results were based on the worst-case scenario, and thus expect that some types of applications will show their reasonable performance on AgentTeamwork especially if we improve GridTcp by reducing its thread context switches.

5.2 User Program Wrapper

We have evaluated the combined performance of the user program wrapper and GridTcp by manually invoking a wrapper at a different cluster node. Each user program wrapper constructs a message routing table, instantiates a GridTcp object, and thereafter invokes the Crypt or the MolDyn benchmark program.

Crypt performs IDEA encryption and decryption on an array of N bytes, based on the master-worker model. Given P computing nodes, the master node divides the array in N/P-byte sub-arrays, each distributed to a different worker. (Note that the master also works as a worker.) Each worker node encrypts its sub-array, then decrypts it, and finally returns the result to the master. The master collects and compares all the results with the original array for validation.

Figure 5 compares the results between the Java and the AgentTeamwork versions that have been executed on 1 to 8 cluster nodes. The both versions did not show a dramatic performance improvement when using two nodes. This is because a half of N bytes had to be exchanged with a worker at once, which obviously neutralized the benefit of parallel execution. The AgentTeamwork version even performed worse in particular for 10M-byte encryption, since GridTcp's bandwidth is only 20% of Java socket. However, as increasing the number of computing nodes toward eight, both Java and AgentTeamwork improved their performance by overlapping communication with computation. The peak performance of Java and AgentTeamwork, (using eight machines) was respectively $3.7 \sim 4.4$ and $2.5 \sim 3.1$ times better than their single execution for all data sizes. Although AgentTeamwork inevitably performs slower than the Java version, its performance penalty can be reduced by scaling up both data size and the number of nodes. In fact,

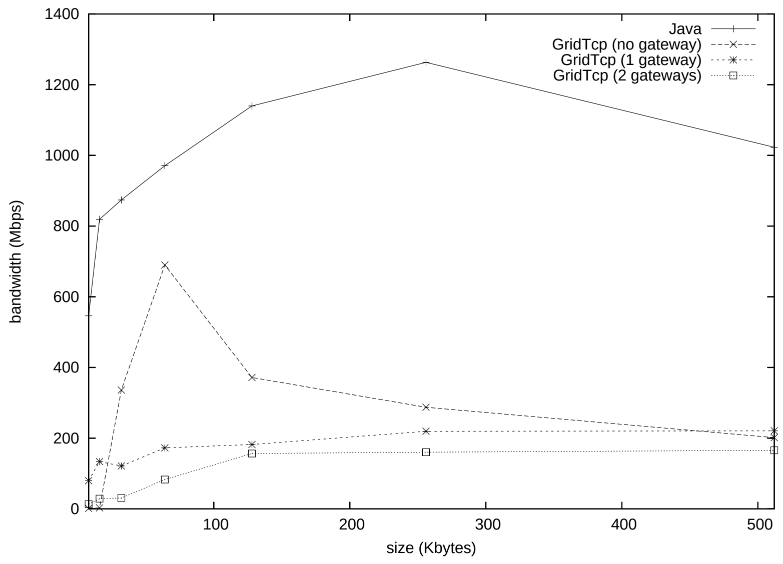


Figure 4: Communication bandwidth.

when we used eight nodes, the ratio of AgentTeamwork to Java in execution time was improved from 1.46 to 1.23 as increasing the data size from 1.25M to 10M bytes.

MolDyn is a molecular dynamics simulation that models N particles interacting under the Lennard-Jones potential in a cubic space. All participating computing nodes repeatedly exchange intermediate results among one another, thus not based on the master-worker model. To be specific, the force on a particle is calculated in a pair wise manner for each of 50 simulation cycles using two nested loops: the outer loop goes over all particles and the inner loop ranges the current particle number to the total number of particles. Given P computing nodes, the outer loop is partitioned in N/P iterations, each calculated by a different computing node. Each node has a copy of all particle data and maintains its consistency with other copies by calling the MPI allreduce( ) function every simulation cycle. This consistency maintenance however accumulates errors in the cubic space as increasing the number of computing nodes, and thus the original MPJ program is designed to indicate errors in validation. We have observed this behavior by running it on our two different cluster systems.

Figure 6 shows the performance results when simulating 2048 and 8788 particles. In the figure, the run time is referred to as the time elapsed for 50-cycle particle simulation only, whereas the total time includes data initialization, distribution, and validation. The AgentTeamwork version partitioned the original program into six functions, each captured in a snapshot file at its end. In terms of 2048-particle simulation, the Java version showed its scalability from 1 to 8 computing nodes, while AgentTeamwork did not demonstrate its competitive performance. Although the computational granularity is diminished with more computing nodes, the exchanged data size is constant. This gave a larger communication penalty to AgentTeamwork. In terms of 8788-particle simulation, both Java and AgentTeamwork demonstrated its scalable performance. Due to MolDyn's pair wise particle computation, increasing particle size exponentially enlarges its computational granularity, which has mitigated AgentTeamwork's performance penalty. Another notable point is that the total time of Java and AgentTeamwork versions is $2 \sim 3\%$ and $5 \sim 15\%$ larger than their run time respectively. This difference is considered of as overhead incurred by AgentTeamwork's execution snapshots.

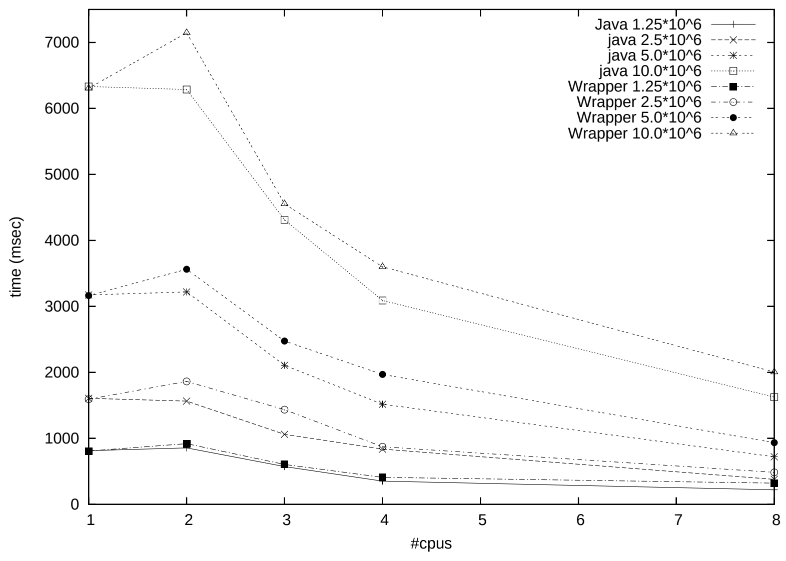


Figure 5: Parallel IDEA encryption.

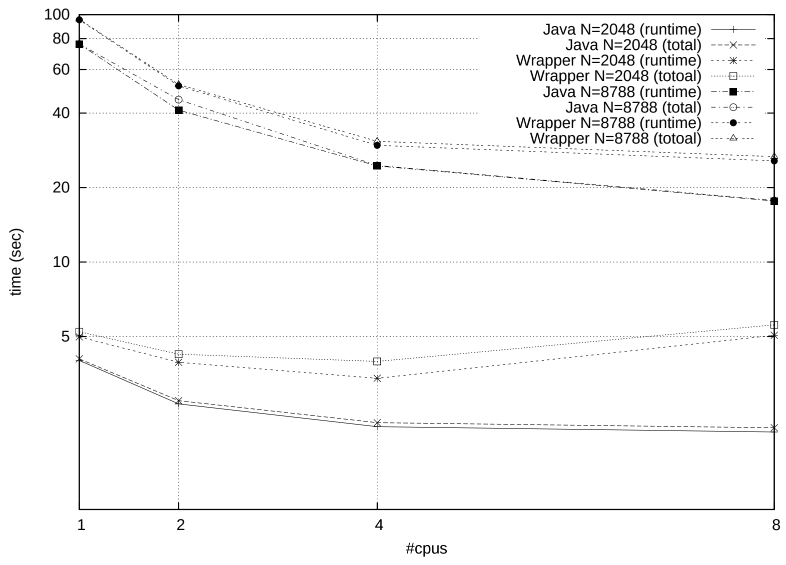


Figure 6: Molecular dynamics performance.

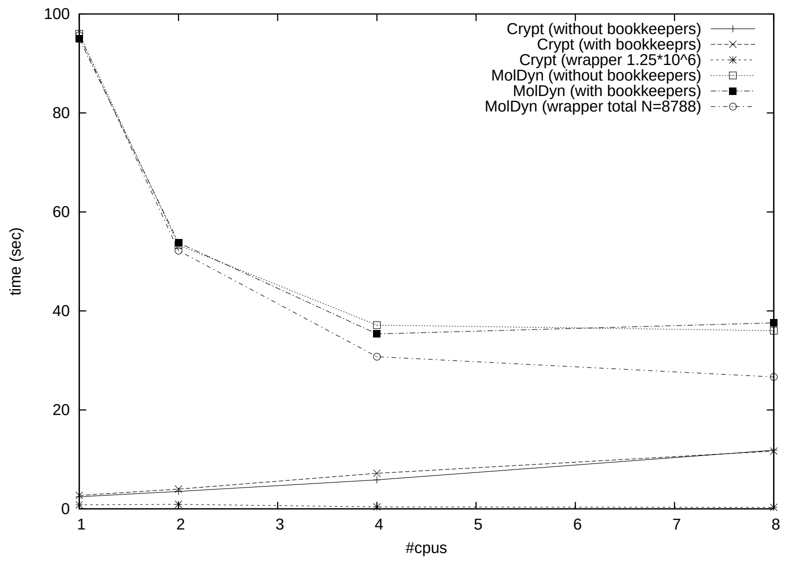


Figure 7: Round-trip performance.

5.3 Round-Trip Performance

We have measured the round-trip time elapsed from a commander agent's invocation to sentinels' termination in AgentTeamwork, using Crypt (1.25M-byte data) and MolDyn (8788 particles). Figure 7 shows the average round-trip time when running these two applications on 1 to 8 cluster nodes five times. In this experiment, a commander agent spawns the same number of sentinels as that of CPUs. To evaluate the overhead incurred by bookkeeper generation, we prepared two test cases: one spawning no bookkeepers and the other with the same number of bookkeepers as that of sentinels. However, our results showed little difference between these two cases. The main reason is hierarchical instantiation of bookkeepers that mitigates the commander's agent-deployment work.

Figure 7 also compares job execution with and without sentinel agents. For the MolDyn program, such difference in the elapsed time increases as proportional to the number of computing nodes: job execution with agents was 1.581, 4.609, and 10.908 seconds slower than direct execution when using 2, 4, and 8 nodes respectively. One of the reasons is the we have not yet modified the sentinel agent to be spawned hierarchically. Furthermore, a commander agent must synchronize with all its sentinels at the end of job execution. These costs depend on the number of sentinels. It is our next task to generate and terminate sentinel agents hierarchically.

From the above results, we feel that AgentTeamwork could demonstrate its competitive performance in both master-worker and non master-worker models, while parallelism and size of each application need to be coarse-grained and large enough to compensate GridTcp and mobile agents' overhead.

6 General Project Management

6.1 Achievements in Year 2003-2004

The PI started the AgentTeamwork project in parallel of his proposal submission to the last year's NSF middleware initiative. He has supervised seven undergraduate students who have been engaged in the AgentTeamwork implementation. This project is constantly attracting top students who have taken any of three senior-level courses the PI is teaching: Operating Systems, Distributed Computing Systems, and Network Design. In fact, three more students will join the project this summer. The following summarizes the undergraduate research achievements with regard to AgentTeamwork.

Period (mo/yr-mo/yr)	Students	Undergraduate research work
01/03 - 06/03	Hyon Kim	designed UWAgent's agent migration.
04/03 - 09/03	Eric Nelson	enhanced UWAgent's agent communication.
04/03 - 06/03	Jon Hagen	designed a computing-resource database.
07/03 - 03/04	Ryan Liu	designed a Xindice interface and the resource agent.
06/03 - 09/03	Doug Kim	designed the sentinel agent's job launching feature.
03/04 - 06/04	Vivian Chan	is porting Java Grande benchmark to AgentTeamwork.
03/04 - 06/04	Tae Suzuki	is re-engineering and documenting all mobile agent code.
06/04 - 09/04	Donya Shirzad	will port MPI applications to AgentTeamwork.
06/04 - 09/04	Shane Rai	will enhance resource-commander agent communication.
06/04 - 12/04	Jon Hendrich	will design a language preprocessor.

The PI has also advised a graduate student who visited from Ehime University, Japan in accordance with its academic exchange program with University of Washington, Bothell. He has played an important role in the system implementation. With his assistance, the PI has prototyped the system and accomplished the preliminary performance evaluation within a year.

6.2 Next Three-Year Plan

To complete the system in three years, the PI is planning the following four task phases: (1) algorithm development, (2) system development, (3) applications development, and (4) dissemination and evaluation. In phase 1, we will design the details of the priority-based distributed scheduling and the execution cryptography proposed in section 2. Phase 2 implements these algorithms in UWAgent and mobile agents running on top of it. We also need to design a language preprocessor. Phase 3 ports onto AgentTeamwork all Java Grande MPJ benchmark programs and several MPI applications developed in the PI's preceding research [9]. Phase 4 conducts the complete evaluation of AgentTeamwork and releases the system as freeware. The PI carries out these phases in collaboration with his research partners of Computer Science Department at Ehime University, Japan, listed below:

1. **Mr. Koichi Kashiwagi:** He was the visiting graduate student from Ehime University. Although he has returned to Ehime and obtained an RA position, we have agreed that we will continue working on this co-project until he obtains a Ph.D., which will take approximately three years.
2. **Prof. Shinya Kobayashi:** Prof. Kobayashi is Mr. Kashiwagi's formal Ph.D. adviser. He brought up the idea of execution cryptography. He has agreed that he will collaborate with me to implement execution cryptography in AgentTeamwork.
3. **Mr. Eric Nelson:** He was one of the UWB undergraduate students working on AgentTeamwork. From this October, he will advance to Ehime University's graduate program and work on execution cryptography under Prof. Kobayashi's supervision.

These collaborators are all self-supported. Therefore, this proposal asks support for the PI's undergraduate students and PI himself. The following summarizes our task assignment and schedule.

	Year 1				Year 2				Year 3			
	2005	2005	2005	2005	2006	2006	2006	2006	2007	2007	2007	2007
Tasks	Win	Spr	Sum	Fall	Win	Spr	Sum	Fall	Win	Spr	Sum	Fall
Algorithm Development												
State capture (Fukuda)	X											
Snapshot maint. (Kashiwagi)	X											
Tcp emulation (Fukuda)	X											
Scheduling (Fukuda/Kashiwagi)	X	X	X	X								
Security (Nelson/Kobayashi)	X	X	X	X	X	X	X	X				
System Development												
Agent platform (UW ugrad)	X	X	X	X	X	X	X	X				
GUI (UW ugrad)									X	X	X	
Commander & resource agents design (Fukuda)	X	X	X	X	X	X	X	X				
Sentinel & bookkeeper agents design (Kashiwagi)	X	X	X	X	X	X	X	X				
Wrapper/Preprocessor (UW ugrad) **Applications Development**	X	X	X	X	X	X						
Applications (UW ugrad)							X	X	X	X	X	
Validation, refinement and results (Fukuda/Kashiwagi)								X	X	X	X	
Dissemination/evaluations												
Seminars and lab work in classes (Fukuda/Kobayashi)		X		X		X		X		X		X
Paper submissions (Fukuda/Kashiwagi)		X		X		X		X		X		X
Annual report (Fukuda)				X				X				
Final project report (Fukuda)												X

6.3 Methods of Evaluation and Dissemination of Results

Results will be exchanged and disseminated through the following methods.

- Exchanges and presentations in research group meetings at UW Bothell and Ehime University.
- Exchanges in the form of presentation and/or demonstrations at the University of Washington's three campuses (Bothell, Seattle, and Tacoma).
- Exchanges, presentations, and pilot use of AgentTeamwork in classes.
- Exchanges, presentations, and demonstrations through involvement in NSF Middleware Initiative, Global Grid Forum (GGF), and IEEE Task Force on Cluster Computing (TFCC)
- Conference papers, presentations and demonstrations with involved undergraduate students as co-authors. Our target conferences include Cluster Computing, CC-Grid (Cluster Computing and the Grid), HPDC (High-Performance Distributed Computing), and MA (Mobile Agents).
- Release of the middleware to the public domain.

7 Summary

This proposal focuses on a group of individual computer owners who wish to mutually offer their own computing resources for their time-critical needs. This type of groups would prefer to a decentralized job management for which we believe mobile agents could serve as an infrastructure. AgentTeamwork deploys commander, resource, sentinel, and bookkeeper agents to orchestrate a user job execution. The PI has prototyped the system with his undergraduate students and collaborators of Ehime University. Our preliminary performance evaluation showed that AgentTeamwork could demonstrate its competitive performance in both master-worker and non master-worker models. The proposal asks support for the PI's undergraduate students and the PI himself to complete the implementation, the evaluation, and the dissemination of AgentTeamwork in the next three years.