THE CHECK-POINTED AND ERROR-RECOVERABLE MPI JAVA LIBRARY OF AGENT TEAMWORK GRID COMPUTING MIDDLEWARE

Munehiro Fukuda, Zhiji Huang

University of Washington, Bothell
Computing and Software Systems
18115 Campus Way NE, Bothell, WA 98033

ABSTRACT

We are implementing a fault-tolerant mpiJava API on top of the AgentTeamwork grid-computing middleware system. Our mpiJava implementation consists of the mpiJava API, the GridTcp socket library, and the user program wrapper, each providing a user with the standard mpiJava functions, facilitating message-recording/error-recovering socket connections, and monitoring a user process. This paper presents the application framework, mpiJava implementation, and communication performance in AgentTeamwork.

1. INTRODUCTION

We are developing the AgentTeamwork grid-computing middleware that deploys Java-based mobile agents over a collection of computing nodes to coordinate parallel execution of Java applications. Of apparent importance is to provide users with high-level communication libraries such as MPI and PVM. However, they do not automatically restore accidental disconnection of communication channels, and therefore do not support migration and resumption of crashed jobs, which becomes a severe problem to grid computing. The reason is the more computing nodes allocated and the more time elapsed, the more chances of node failure a job may face during the course of its computation. In addition, to accelerate its execution speed, a job needs to migrate to a less-loaded node at run time.

As a simple solution, some middleware systems set their target on embarrassingly parallel or bag-of-task applications that require inter-process communication only at the beginning and the end of execution, namely for their parameter distribution and result collection [1]. However, they obviously narrow down application domains. Another solution is to use FT-MPI [2], a fault-tolerant MPI that restarts a crashed process and restores its MPI communicator, although an application is still responsible to recover messages lost in transit to a crashed process. Furthermore the process must resume from its top of computation.

To cope with these problems, it is essential to repeat taking a snapshot of computation and to resume a process from the latest snapshot. The Condor MW project is such a system that has focused on the master-worker model in PVM [3]. Its MW library in a master process keeps track of and retrieves connections to all its worker processes, yet inter-worker communication is not retrievable due to the lack of snapshots taken at each worker.

An enhanced solution is to replace MPI's underlying TCP connection with the Rock/Rack reliable TCP [4], using which a user process can reestablish a new TCP connection to its "resumed" peer process, provided it is called back from and given an IP address of this peer process. This, however, forces users to develop mobile-aware applications.

Based on the above background, we are implementing a check-pointed and error-recoverable Java version of MPI on top of the AgentTeamwork middleware. While complying with the original mpiJava's API, our implementation uses Java sockets we have modified so as to save a history of old TCP messages, to restore disconnected TCP connections, and to deliver lost messages to resumed processes.

This paper introduces the overview of AgentTeamwork, presents the implementation techniques for our fault-tolerant mpiJava, and discusses its communication performance.

2. PROGRAMMING INTERFACE

Using mobile agents, AgentTeamwork coordinates job execution over a collection of desktop/cluster computing nodes, each owned by a different user but registered in a common ftp server, (e.g., ftp.tripod.com in our implementation). To participate in a grid-computing community, a user downloads XML-described resource and user-account files as well as the AgentTeamwork software kit, and thereafter runs the UWAgent mobile-agent execution platform on his/her computing node in background so as to exchange user jobs with other community members.

AgentTeamwork does not need a central cycle server for job submission and coordination. Each user can inde-

This work is fully funded by National Science Foundation's Middleware Initiative (No.0438193).

pendently submit a job with a commander agent, one of mobile agents provided by AgentTeamwork. This agent starts a resource agent that searches its local XML files for the computing nodes best fitted to the user job's resource requirements. Receiving such candidate nodes, the commander agent spawns a sentinel and a bookkeeper agent. These agents hierarchically spawn as many descendants as the number of nodes required by the job execution. Each sentinel launches a user process and repeats sending its execution snapshot to the corresponding bookkeeper agent. Upon an agent crash, its parent or child agent resumes the crashed agent with the latest snapshot retrieved from the bookkeeper. All results are forwarded through the agent hierarchy from the bottom to the commander agent that thereafter reports them to the client user through the monitor, by email, or by files.

Figure 1 shows the AgentTeamwork execution layers from the top application level to the underlying operating systems. The system facilitates the mpiJava API for Java applications, while they may call native functions and use socket-based communication as a matter of course. Complying with the original API, AgentTeamwork distinguishes two versions of mpiJava implementation: one is *mpiJava-S* that establishes inter-process communication using the conventional Java socket, and the other is *mpiJava-A* that realizes message-recording and error-recoverable TCP connections using our *GridTcp* socket library. The implementation is user-selectable with the type of arguments passed to the *MPI.Init( )* function. Below mpiJava-S and mpiJava-A is the user program wrapper that periodically serializes a user process into a byte-streamed snapshot and passes it to the local sentinel agent. As described above, the sentinel agent sends every new snapshot to its corresponding bookkeeper for recovery purposes. All these agents are executed on top of the UWAgent mobile agent execution platform.

<table>
<tr><td colspan="2">Java user applications</td></tr>
<tr><td colspan="2">mpiJava API</td></tr>
<tr><td>mpiJava-S</td><td>mpiJava-A</td></tr>
<tr><td>Java socket</td><td>GridTcp</td></tr>
<tr><td colspan="2">User program wrapper</td></tr>
<tr><td colspan="2">Commander, resource sentinel, and bookkeeper agents</td></tr>
<tr><td colspan="2">UWAgent mobile agent execution engine</td></tr>
<tr><td colspan="2">Operating systems</td></tr>
</table>

Fig. 1. AgentTeamwork execution layer

An application program is coded in the AgentTeamwork-specific template as shown in Figure 2. If the program uses mpiJava-A, it must include a GridTcp object in a declaration of system-provided members (line 4). The code consists of a collection of methods, each named *func_* plus an index starting from 0 and returning the index of the next method to call. The application starts from *func_0*, repeats calling a new method indexed by the return value of its previous method, and ends in the method whose return value is -2, (i.e., *func_2* in this example). The *MPI.Init* function invokes mpiJava-A when receiving an ipEntry object that is initialized by the user program wrapper to map an IP name to the corresponding MPI rank (line 8). Following *MPI.Init*, a user may use any mpiJava functions for inter-process communication (lines 13, 14, 16, and 22). The user program wrapper takes a process snapshot at the end of each function call. Since GridTcp maintains old MPI messages internally, a process snapshot also contains these messages in it. When the application is moved to or resumed at a new computing node, GridTcp retrieves old messages from the latest snapshot and resends them if they have been lost on their way. We will ultimately relieve users from this framework-based programming by implementing a a JavaCC/ANTLR-based language preprocessor that automatically partitions their Java applications into a collection of it func_ methods.

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

Fig. 2. AgentTeamwork-specific code template

3. IMPLEMENTATION TECHNIQUES

This section details the underlying layers that support our mpiJava implementation such as the MPJ package, the GridTcp library, the user program wrapper, and the sentinel agent.

3.1. MPJ Package

This is a collection of Java programs that interfaces user applications with the underlying Java socket or GridTcp socket

communication. It includes:

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

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

3.2. GridTcp

GridTcp provides message-recording and error-recoverable TCP connections. It saves each connection's incoming and outgoing messages in its internal queues. The queues are captured 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 the underlying user program wrapper takes a new snapshot.

Figure 3 gives an example where two processes are engaged in the same MPI application, running at n1 and n2 of the *uwb.edu* domain, and identified with rank 1 and 2 respectively. When a process migrates from n2 to n3, their TCP connection is broken, which thus asserts an exception to the underlying layers, (i.e., the user program wrapper and the sentinel agent). The sentinel agent with rank 1 receives a new IP address from the one with rank 2, (i.e., n3). Thereafter, these two sentinels pass the rank2/n3 pair to their user program wrapper that has GridTcp restore in-transit messages from the latest snapshot, update its routing table, and reestablish the previous TCP connection.

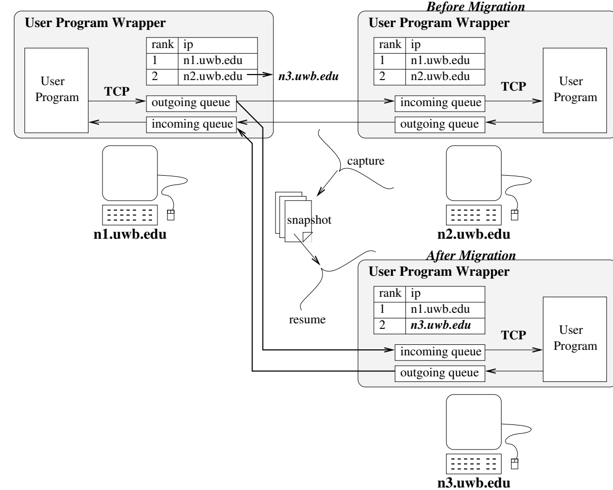

Fig. 3. TCP maintenance upon migration.

3.3. User Program Wrapper

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

This solution, however, burdens users with framework-based programming. We will address this burden by extending the UCI Messengers' compiler technique that converts a user program into a series of functions [5]. Figure 4 shows the simplest check-pointing example where nine sequential statements are partitioned into three functions, named func_0, func_1, and func_2, each returning the index of the next function to call. The user program wrapper saves this index value as well as all the user data members, and thereafter calls the indexed function. Upon a process crash, the wrapper retrieves the last index from the corresponding snapshot and resumes the process from the indexed function. The preprocessing technique for more complicated programs is described in [5].

3.4. Sentinel Agent

The sentinel agent is one of the AgentTeamwork-provided mobile agents. Dispatched to a different computing node, each sentinel launches a user program wrapper as passing a user program name and its parameters to the wrapper. Every five seconds the sentinel checks if the wrapper has taken a new execution snapshot and sends it to the corresponding

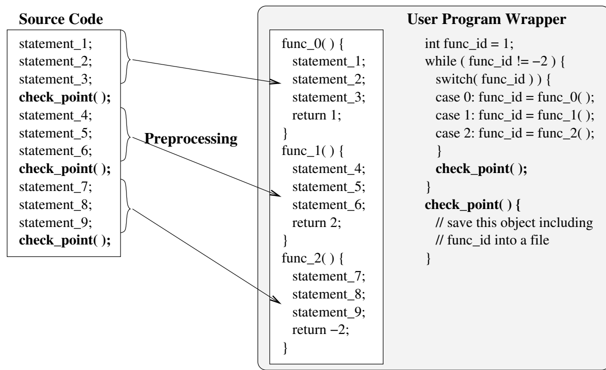

Fig. 4. Source program preprocessing.

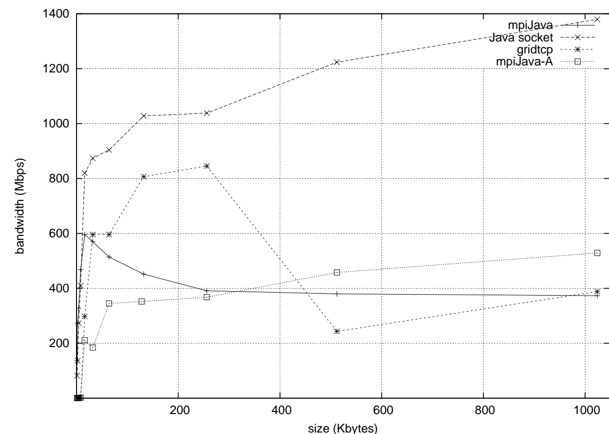

Fig. 5. Communication performance.

bookkeeper if it is new. When a sentinel is resumed after a crash or a migration, it multi-casts a pair of its new IP name and rank to its parent and child agents that further forward their ascendants and descendants. Upon receiving such a new IP/rank pair, each agent reinitializes its user program wrapper that directs GridTcp to recover broken connections.

4. COMMUNICATION PERFORMANCE

We have implemented the basic features of the AgentTeamwork system including 14 major functions of mpiJava such as *MPI.Send( )*, *Recv( )*, *Bcast( )*, *gather( )* and *scatter( )*. Using Java Grande MPJ benchmark's PingPong program [6], we have compared the communication performance among four different executions: (1) mpiJava: the program executed with the original mpiJava, (2) Java: the one converted to use Java sockets and executed with JVM, (3) GridTcp: the one converted to use GridTcp sockets and executed on AgentTeamwork, and (4) mipJava-A: the one executed on top of mpiJava-A and AgentTeamwork. We conducted our evaluation on two cluster systems: one is a Myrinet-2000 cluster of eight computing nodes, each with a 2.8GHz-Xeon processor, and the other is a Giga-Ethernet cluster of 24 nodes, each with a 3.2GHz-Xeon processor. Note that all processors have 512MB system memory.

Figure 5 compares the performance of these four versions on the Myrinet cluster. Java demonstrated its ideal network bandwidth. GridTcp marked only 20% $\sim$ 80% of the Java version due to its message-recording overhead. The original mpiJava showed its 400Mbps constant bandwidth, whereas mpiJava-A gradually increased its bandwidth and even showed higher bandwidth than the original mpiJava, (i.e., 528.916Mbps at 1024K-byte message transfer). For our evaluation on the Giga-Ethernet cluster, we received the similar results where mpiJava-A performed 381.944Mbps at 1024K-byte transfer, whereas the original mpiJava showed slightly lower bandwidth, (i.e., 304.155Mbps).

5. CONCLUSIONS

This paper has discussed the design, the implementation, and the preliminary performance of mpiJava that facilitates check-pointed and error-recoverable message passing on top of the AgentTeamwork grid-computing middleware system. The paper has also demonstrated that mpiJava-A's performance is competitive to that of the original mpiJava library.

Our next milestone is to complete our implementation of mpiJava-S and mpiJava-A APIs and to design a language preprocessor that automatically partitions a user program into a collection of check-pointed functions.

6. REFERENCES

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

[2] FT-MPI: HARNESS Fault Tolerant MPI, "http://icl.cs.utk.edu/ftmpi/," 2003.

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

[4] V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proc. MOBICOM'02*, Atlanta, GA, September 2002, pp. 95–106, ACM Press.

[5] C. Wicke, L. Bic, M. Dillencourt, and M. Fukuda, "Automatic state capture of self-migrating computations in MESSENGERS," in *Proc. MA'98*. September 1998, pp. 68–79, Springer.

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