THE DESIGN CONCEPT AND INITIAL IMPLEMENTATION OF AGENT TEAMWORK GRID COMPUTING MIDDLEWARE

*Munehiro Fukuda**

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

Koichi Kashiwagi, Shinya Kobayashi

Ehime University
Department of Computer Science
3 Bunkyo, Matsuyama, Ehime 790, Japan

ABSTRACT

AgentTeamwork is a grid-computing middleware system that dispatches a collection of mobile agents to coordinate a user job over remote computers in a decentralized manner. A parallel-computing job is check-pointed periodically, moved by a mobile agent for better performance, and resumed upon a crash. The system also restores broken communication with its error-recoverable socket and mpiJava libraries. This paper presents AgentTeamwork's design concept, implementation, and basic performance.

1. INTRODUCTION

While grid computing has drawn an emergent attention in various applications, it has not yet obtained a successful popularity among all end users. One reason is considered as a centralized style of resource and job management, widely adopted by many grid-computing middleware systems [1]. In spite of its simplicity, centralized middleware suffers from two restrictions: (1) a powerful central server is essential to manage all slave computing nodes, and (2) applications are inevitably based on the master-slave or bag-of-task programming model. Therefore, such middleware may not sufficiently benefit those who can not access a shared cycle server or who want to develop their applications with various patterns of inter-process communication. One extreme example is common ISP users who may wish to mutually offer their desktop computers for grid computing but are obviously unable to share a public cycle server.

For the last decade, mobile agents have been highlighted as a prospective infrastructure of decentralized systems for information retrieval, e-commerce, and network management [2]. In fact, several grid-computing systems have been proposed to use mobile agents as their infrastructure, claiming the merits of mobile agents such as their navigational autonomy used for automated resource search, their state-capturing capability for job migration, and their inherent parallelism for faster job execution. Regardless of these merits, mobile-agent-based systems have not yet outperformed commercial grid-computing middleware. This is because they have not completely departed from the master-worker model and thus used their mobile agents for a job deployment and a result collection. Therefore, mobile agents could not give more than an alternative approach to grid-computing middleware implementation.

Contrary to the above approach, we apply mobile agents to a decentralized coordination of user jobs not necessarily fitted to the master-worker model. Our only assumption (and restriction) is that grid-computing users can share at least one ftp server to register their own computing resources. Once a user downloads resource information from the ftp server, his/her mobile agents are responsible to regularly probe the workload of remote machines, to dispatch a job to the best fitted machines, to monitor and checkpoint the job execution, to migrate a job to a lighter loaded machine, and to resume a crashed job from the latest snapshot at a new machine. Those agents engaged in the same job form a team to pursue this distributed job coordination. We have implemented this agent collaboration in the AgentTeamwork grid-computing middleware system.

This paper presents the overview, the internal design, and the basic performance of the AgentTeamwork system.

2. SYSTEM OVERVIEW

AgentTeamwork targets a computational community agreed on by remote desktop/cluster computer owners and formed with a common ftp server, (e.g., ftp.tripod.com in our implementation). Such an ftp server is used only to store a collection of XML-based user account/resource files and AgentTeamwork's software kit. The latter includes the *UWAgent* mobile agent execution platform, an eXist database manager, all mobile agent programs necessary to coordinate a user job, and Java packages to be imported when developing an application. Once each computing node downloads these ftp contents, it runs the *UWAgent* execution platform

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

in background so as to dispatch and to accept user jobs.

An application is coded in the AgentTeamwork-specific template as shown in Figure 1. It defines system-initialized variables, one of which is GridTcp, our Java socket library capable of restoring disconnected TCP connections and delivering lost messages to their destination processors. The user program 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. Given a string array, 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 that returns -2, (i.e., *func_2* in this example). The system takes a computation snapshot at the end of each function call, so that the application can be resumed at a new computing node from the latest snapshot upon its migration or accidental crash. Programmers can also use mpiJava whose API complies with the original version but whose implementation uses GridTcp to make their applications mobile to and recoverable at a new node. We will ultimately relieve users from this framework-based programming by implementing a language preprocessor that partitions their Java applications into a collection of *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 rank
7     public int func_0(String args[]){// constructor
8       .....;        // actual statements inserted
9       return 1;     // call func_1( )
10    }
11    public int func_1( ) { // called from func_0
12      .....;        // actual statements inserted
13      return 2;     // call func_2( )
14    }
15    public int func_2( ) { // the last function
16      .....;        // actual statements inserted
17      return -2;    // application terminated
18  } }
```

Fig. 1. AgentTeamwork-specific code template

Figure 2 shows a series of job coordination in AgentTeamwork. A user job is submitted with a commander agent, one of the system-provided mobile agents. This agent spawns a resource agent that launches an eXist database, searches for XML files best fitted to the user's resource requirements, returns an initial itinerary of available computing nodes to the commander, and keeps alive to periodically probe the workload of remote nodes. Given that node itinerary, the commander agent spawns a sentinel and a bookkeeper agent. These agents spawn as many descendants as the number of nodes required by the job execution. Each sentinel migrates to a different 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. We use the following formulas F1 $\sim$ F3 to assign a processor id to each agent and to map a sentinel to the corresponding bookkeeper.

F1: $rank = sId - (2 + (2N-3)\sum_{i=0}^{\frac{\ln(sId/2)}{\ln(N)}-1} N^i)$
where $rank$ = a process id, sId = a sentinel id, and N = # children each agent can spawn.

F2: $rank = bId - (3 + (3N-4)\sum_{i=0}^{\frac{\ln(bId/3)}{\ln(N)}-1} N^i)$
where bId = a bookkeeper id.

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

At a different node, each bookkeeper with $rank = i$ waits for and reroutes a new snapshot to its two neighbors with their $rank = i-1$ or $i+1$, anticipating its accidental crash. Each agent monitors the local resource conditions and migrates to another node listed in its itinerary, once the current node no longer satisfies its resource requirements. It also takes charge of monitoring and resuming its parent/child agents from their latest snapshot if they are crashed. If agents exhaust their node itinerary, the commander requests the resource agent to make another itinerary. Upon a job termination, the standard outputs are all returned as a string array to the commander that then passes them to the client user through his/her display, by email, or by files.

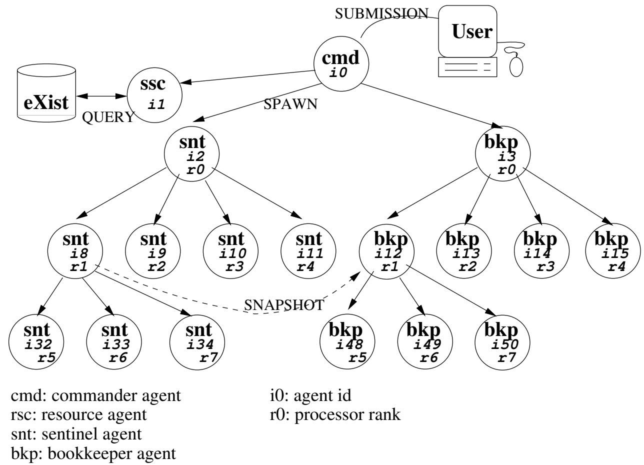

Fig. 2. Job coordination made in a agent hierarchy

3. INTERNAL DESIGN

This section discusses AgentTeamwork's agent execution platform and job migration/resumption algorithm.

3.1. Mobile Agent Execution Platform

Java is most widely used to implement mobile agents because of its architectural independence, object serialization,

and runtime class loader. Although various Java-based mobile agents have been released to the public [2], we have designed a new platform named *UWAgent*, since we need grid-computing-oriented features in agent naming and communication, agent termination, and runtime job execution.

For naming and communication, *UWAgent* uses a concept of agent domain. Submitted from the Unix *shell*, an agent forms a new agent domain identified with a triplet (IP address, a time stamp, and a user name), becomes the domain root with id 0, and receives N, the maximum number of children each agent can spawn. The root agent can create $N-1$, and each of its descendants can further generate N children. When a new child is spawned from an agent i, it receives $id = i * N + seq$ where $seq =$ an integer starting from 0 if $i \neq 0$ or 1 if $i = 0$. This naming scheme enables us to use the formulas defined in Section 2 to map a sentinel to a bookkeeper as well as allocate a processor rank to them.

For termination, *UWAgent* permits an agent to wait for the termination of all its children or even to kill them with one method. This feature helps AgentTeamwork terminate all descendants of a commander agent, (i.e., a root agent) when its client user's application has been finished or aborted.

For runtime job execution, *UWAgent* executes user jobs as Java threads. Contrary to most mobile-agent systems that launch jobs as Unix processes, *UWAgent* strictly schedules user jobs with Java thread's suspend and resume method, based on their runtime attributes such as the number of migrations, the total CPU time, and the execution priority.

3.2. Job Migration and Resumption

Using the following algorithms, AgentTeamwork moves a job to an idle node or resumes a crashed job at a new node.

Job migration is achieved by a sentinel that chooses a new node from its itinerary and migrates there with its user process. This however breaks all TCP connections to the process. Other processes that have detected any broken connections assert an exception to their user program wrapper that makes its sentinel wait for a "restart" message to come from the migrating sentinel, restore the wrapper with the new location information, and finally resume the process.

Job resumption is more difficult, since a job has been crashed with its sentinel agent. Two resumption scenarios are considered: one for resuming a child sentinel and the other for a parent sentinel. As shown in Figure 3-A, a child sentinel is resumed through the following five steps: (1) a parent sends a "ping" message to all its children every five seconds; (2) if detecting a child crash, it sends a "search" message to the corresponding bookkeeper; (3) the parent receives a "retrieve" message including the crashed sentinel's snapshot from the bookkeeper; (4) the parent sends this snapshot to a new node's *UWAgent* platform that resumes the crashed sentinel; and (5) the resumed sentinel sends a "restart" message to all the other sentinels that have waited for broken connections to be restored. Figure 3-B shows a parent sentinel resumed in five steps; (1) a child sentinel receives a "ping" message from its parent every five seconds; (2) if receiving no ping for $5 \times agent_id$, it concludes that all its ancestors are dead and thus sends a "search" message to the corresponding bookkeeper; and (3) through to (5) are the same as those in the child resumption scenario. The parent resumption needs a long period of time to detect a parent crash in the step 2. If all the ancestors of a given agent are crashed, this overhead must be unnecessarily repeated until the root agent is resumed. As an improvement, we allow a resumed parent to assume that its ancestors are all crashed, thus to skip the step 2, and to resume its parent immediately.

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

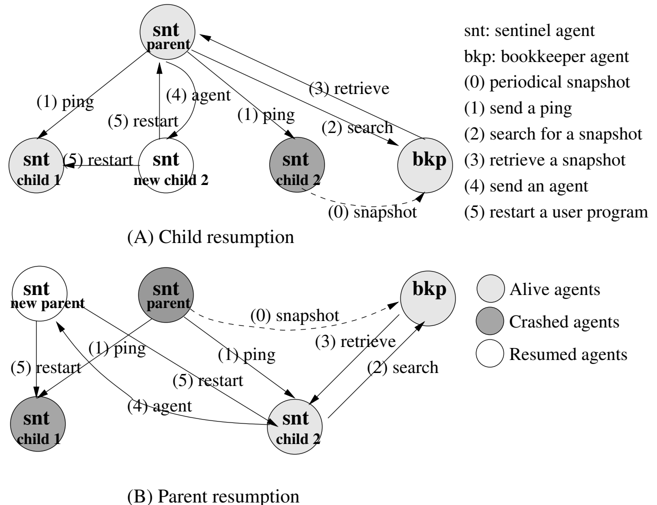

Fig. 3. Agent resumption

4. PERFORMANCE EVALUATION

We have implemented the basic features of the AgentTeamwork system, while still developing the mpiJava API and job-scheduling algorithms. To evaluate its performance, we used the following three Java Grande MPJ benchmark programs [3]: (a) Series: a Fourier coefficient analysis, (b) RayTracer: a 3D ray tracer, and (c) MolDyn: a molecular dynamics simulation. We have converted them into three versions: (1) Java version: the original programs using ordinary Java sockets, (2) GridTcp version: the ones using the GridTcp sockets, (3) AgentTeamwork version: the GridTcp version coordinated by AgentTeamwork. They were further modified to keep running 10 times, each taking an execution snapshot in both the GridTcp and AgentTeamwork versions.

The evaluation used a Myrinet-2000 cluster of eight nodes, each with 2.8GHz Pentium-4 Xeon and 512MB memory.

Figure 4 shows a performance of Series that computes a different block of 100,000 coefficients with each processor and collects all results at the rank 0 processor. Since this is a master-slave model and its data size is small, AgentTeamwork demonstrated its very competitive performance.

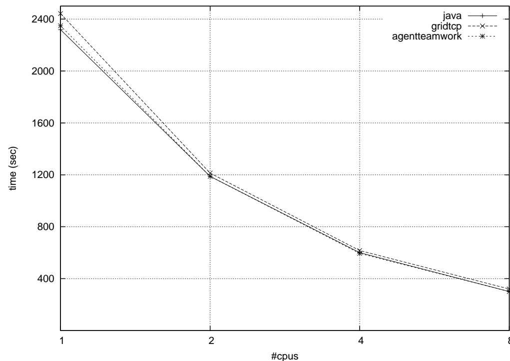

Fig. 4. Performance of Series: Fourier coefficient analysis.

Figure 5 shows a performance of RayTracer that renders a given scene at 500×500 pixels in parallel, calls *MPI.allreduce* to compute a global checksum, and collects results at the rank 0. Although this is a master-slave program, its large data size incurs substantial overheads in communication and check-pointing. Therefore, AgentTeamwork performed $4.4\% \sim 18.6\%$ slower than the Java version.

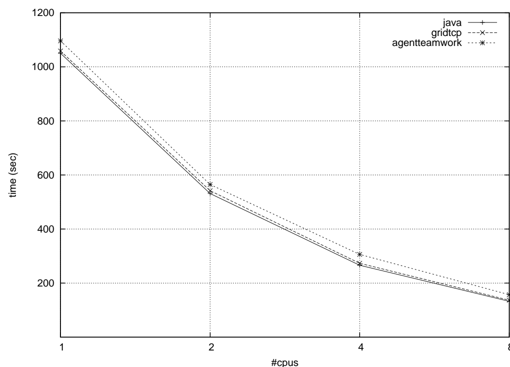

Fig. 5. Performance of RayTracer: 3D ray tracer.

Figure 6 shows a performance of MolDyn that models 8,788 particles interacting under the Lennard-Jones potential in a cubic space, calculates a particle force for 50 cycles, and exchanges the space information among processors every cycle. Since this is not based on the master-slave model and needs cyclic communication, AgentTeamwork performed 1.3 to 2.9 times slower than the Java version.

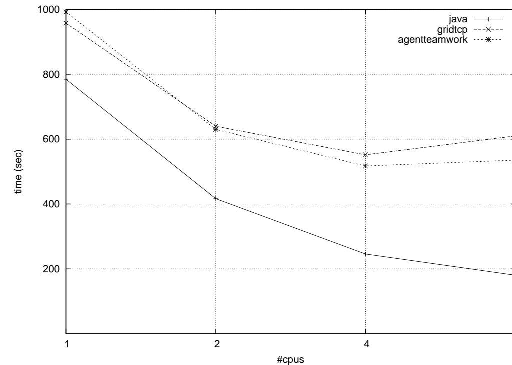

Fig. 6. Performance of MolDyn: Molecular dynamics.

Table 1 summarizes the ratio of agent-resumption time over total execution time, where a sentinel agent is intentionally crashed and thereafter resumed at a new node. AgentTeamwork has successfully restrained its resumption overhead to at most 7% of the entire computation.

#CPUs	Series	RayTracer	MolDyn
1	1.00%	0.51%	1.02%
2	0.28%	3.79%	1.34%
4	2.53%	7.00%	2.65%

Table 1. Overhead of process resumption

5. CONCLUSIONS

In this paper, we have presented the design concept and internal implementation of AgentTeamwork. The system demonstrated its scalable and competitive performance when running three Java Grande MPJ benchmark programs.

Our next step is to complete and enhance AgentTeamwork's programming support, job scheduling algorithms, and security to make the system robust for practical use.

6. REFERENCES

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

[2] Dejan Milojicic et al., **Mobility** *Processes, Computers, and Agents*, Addison-Wesley, Reading, MA, 1999.

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