Fault-Tolerant Job Execution over Multi-Clusters Using Mobile Agents [*]

Munehiro Fukuda Emory Horvath Solomon Lane

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011
{*mfukuda, emoryh, solomonl*}*@u.washington.edu*
Phone: 1-425-352-3459 Fax: 1-425-352-5216

Abstract

AgentTeamwork is a mobile-agent-based job coordination system that targets a mixture of computing nodes, some directly connected to the public Internet and others simply clustered in a private IP domain but not managed by a commodity job scheduler. The system allows its mobile agents to carry a user job with them from the public to private IP domains as well as to form a hierarchy where agents are recursively spawned to launch a job at a different node, to monitor their parent and children, to resume them upon their crash, and to relay a job-termination signal to the root agent, (i.e., the one directly communicating with a user). To manage multiple clusters, all agents running within the same cluster constitute a subtree derived from the agent residing at their cluster head. This algorithm enables mobile agents to deploy a job to multiple cluster heads and henceforth to their cluster-internal computing nodes, as well as to monitor and resume the job both across clusters and within each cluster. This paper presents our agent-based algorithm and its performance for job deployment, monitoring, and resumption over multiple clusters.

Keywords: Job coordination, fault tolerance, grid middleware, mobile agents

1 Introduction

Job execution over multiple clusters has rapidly attracted users in grid computing as one of the most cost-effective methods to scale up processor parallelism and thus to increase computing power. The challenge of multi-cluster job execution includes not only the study of efficient co-scheduling algorithms, (particularly those aware of inter-cluster network bandwidth [14, 2]) but also development of infrastructures to facilitate fault-tolerant job execution over multiple clusters.

Most conventional infrastructures are based on a centralized two-level job deployment where a central job co-allocator orchestrates multiple remote clusters, each mandated by a commodity cluster-local scheduler such as GRAM and PBS [5, 13]. Contrary to its simplicity, this approach would encounter the following structural, performance, and fault-tolerance problems: (1) all cluster nodes are not systematically managed by their cluster-local scheduler and moreover turned on and off asynchronously as typically seen in instructional computers; (2) the job-deployment latency increases in linear to the number of independent clusters; (3) Fault recovery can be no longer handled within each cluster for a job whose inter-process communication takes place over multiple clusters, which in turn means that a central job co-allocator must be furnished with comprehensive fault-tolerant intelligence.

To address these problems, we have developed a mechanism for fault-tolerant multi-cluster job execution that uses a hierarchy of mobile agents, each deployed to a different cluster head or gateway where it further deploys a hierarchy of child agents to cluster-internal nodes in $O(logN)$. Each agent launches a user job, takes its on-going execution snapshots, passes them to another agent running on a different cluster, monitors their parent and child agents, and resumes them upon a crash. We have implemented this mechanism in the AgentTeamwork system [10]. It is our main focus to present how AgentTeamwork can address the structural, performance, and fault-tolerance problems in multi-cluster job execution as well as to analyze its competitive performance.

The rest of the paper is organized as follows: Section 2 gives an overview of AgentTeamwork; Section 3 explains our implementation of fault-tolerant multi-cluster job execution; Section 4 analyzes AgentTeamwork's performance in job deployment and check-pointing as well as compares

[*] This research is being conducted with full support from the National Science Foundation's Middleware Initiative (No.0438193).

AgentTeamwork and Globus; Section 5 looks at related work; and Section 6 concludes our discussions.

2 AgentTeamwork

AgentTeamwork is grid-computing middleware that supports fault-tolerant job execution over multiple computing nodes, using an hierarchy of mobile agents [10]. The system distinguishes them in commander, resource, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, job deployment and monitoring, and job-execution bookkeeping respectively. These mobile agents are exchanged among remote computing nodes, each running a UWAgents mobile-agent execution platform [11].

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 retrieves appropriate XML-described resource information from its local XML database as well as downloads new information from a shared ftp server if necessary, (*ftp.tripod.com* in our current implementation).

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 GUI.

A user program is wrapped with a user program wrapper, one of the threads running within a sentinel agent. The wrapper provides a user program with error-recoverable TCP and file libraries, each named *GridTcp* [9] and *GridFile*, which serializes its execution snapshot in bytes including in-transit messages and file contents. The system also facilitates a fault-tolerant version of the mpiJava API [16] that has been actually implemented with GridTcp.

Since Java does not serialize a program counter, it is impossible to resume a user program where it has been suspended or crashed. To allow a program to restart from its middle, AgentTeamwork defines its programming framework as shown in Figure 1. The code consists of a collection of methods, each of which is named *func_* appended by a 0-based index. An application starts from *func_0* (line 5), repeats calling a new method indexed by a return value of the current method (lines 10 and 14), and ends in the method whose return value is -2 (line 17). The user program wrapper takes a process snapshot at the end of each function call, so that the process can resume its computation from the last function call.

```
 1  public class MyApplication {
 2    public int funcId;                  // used by system
 3    public GridTcp tcp;                 // used by GridTcp
 4    public GridIpEntry ipEntry[];       // used by GridTcp
 5    public int func_0(String args[]){   // constructor
 6      MPJ.Init( args, ipEntry );        // invoke mpiJava
 7      .....;                            // more statements
 8      return 1;                         // calls func_1
 9    }
10    public int func_1( ) {              // from func_0
11      .....;                            // more statements
12      return 2;                         // calls func_2
13    }
14    public int func_2( ) {              // from func_1
15      .....;                            // more statements
16      MPJ.finalize( );                  // stops mpiJava
17      return -2;                        // terminated
18  } }
```

Figure 1. An AgentTeamwork's user code with function-based snapshots

As shown in Figure 2, AgentTeamwork also allows user-initiated check-pointing operations where a Java application starts from *main( )* as usual (line 9), instantiates and registers local objects to save in execution snapshots (lines 16-17), takes an execution snapshot at any point of time (line 7), and resumes its objects from the latest snapshot (lines 12-13).

```
 1  import AgentTeamwork.Ateam.*;
 2  public class MyApplication extends AteamProg {
 3    public MyApplication(Object o){} // reserved
 4    public MyApplication( ) {}       // user own
 5    private void compute( ) {       // user computation
 6      ...;
 7      ateam.takeSnapshot(phase);    // check-pointing
 8    }
 9    public static void main( String[] args ) {
10      MyApplication program = null;
11      if ( ateam.isResumed( ) ) {// resumption
12        program = (MyApplication)
13           ateam.retrieveLocalVar( ‘‘program’’ );
14      } else {                      // initialization
15        MPI.Init( args );           // invoke mpiJava
16        program = new MyApplication( );
17        ateam.registerLocalVar( ‘‘program’’, program );
18      }
19      program.compute( );
20      MPI.Finalize( args );
21  } }
```

Figure 2. An AgentTeamwork's user code with user-initiated snapshots

3 Job Coordination over Multi-Clusters

Figure 3 shows AgentTeamwork's hierarchical job deployment in a single address domain, which brings the fol-

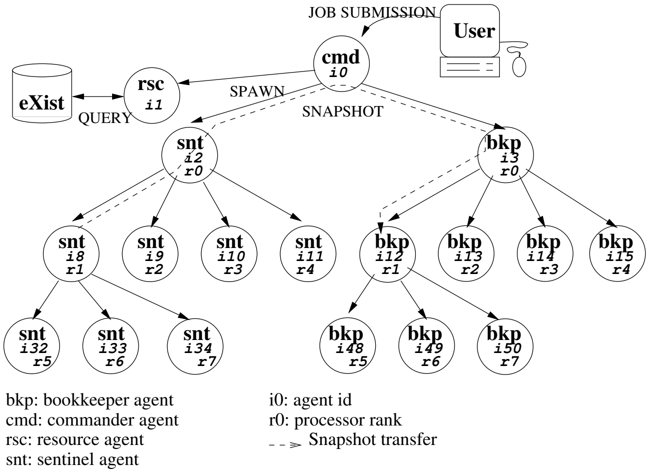

Figure 3. AgentTeamwork's original algorithm for job coordination

lowing four benefits: (1) a job is deployed in a logarithmic order; (2) each agent computes its tree-unique identifier (simply abbreviated to *id* in the following discussions) without a central name server [1], with which an agent computes an MPI rank for its user process; (3) each sentinel monitors its parent and child agents as well as resumes them upon a crash in parallel; and (4) each user process is check-pointed by a different sentinel and its snapshot is maintained by a separate bookkeeper, which improves snapshot availability and thus enhances fault tolerance.

Despite those merits, this algorithm cannot be re-used directly for inter-cluster job coordination. In the rest of this section, we first examine challenges in this coordination work and thereafter show a better approach to applying our hierarchical algorithm to multiple clusters.

3.1 Challenges

There are four challenges in applying an agent hierarchy to multi-cluster job execution.

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

The second is how to send to and maintain at bookkeepers a collection of execution snapshots that have been taken across multiple clusters. A pair of sentinel and bookkeeper agents should not reside at the same computing node and moreover in the same cluster for increasing snapshots' availability. Of importance is to consider an efficient manner to pass snapshots from each sentinel in a cluster-private domain to the corresponding bookkeeper in the public address domain without causing congestion.

The third is where to resume a crashed cluster node and furthermore a crashed cluster gateway. If there are no more available nodes in a cluster to resume a job or if a cluster gateway itself has crashed, AgentTeamwork must search for a new cluster to resume a crashed job. The main problem is that two or more sentinels, (thus user processes) may be running in the previous cluster. Therefore, we need to migrate all of them at once to the new cluster.

The fourth is how to establish and resume inter-process communication over multiple clusters, each constituting a private address domain. Related to the third problem, all user processes in the same cluster may need to migrate to a different cluster upon a crash of their cluster gateway. Therefore, inter-process communication must be not only maintained but also resumed through cluster gateways.

Our solution to each of these challenges includes: (1) the system uses a two-level agent hierarchy, where the outer hierarchy is allocated to cluster gateways while the leftmost agent at each tree level creates an inner tree for a cluster whose gateway is managed by this agent's parent; (2) all snapshots taken inside a cluster are directly passed from its gateway to the corresponding bookkeeper; (3) an agent running at a cluster gateway resumes all its descendants at a new cluster whereas all the other sentinels terminate upon detecting their gateway's crash; (4) GridTcp has been enhanced to capture inter-cluster TCP messages in its execution snapshot for an anticipated cluster recovery. In the following, we will explain the details of our solution.

3.2 Job Deployment

Figure 4 illustrates AgentTeamwork's inter-cluster job deployment. It distinguishes clusters from independent desktops by grouping them into the left and right subtree 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. 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 as *computing-node agents*.

Each agent is given the same arguments upon its instantiation: a list of cluster gateway names, a list of their cluster sizes and a list of computing node names below each gateway. Given these arguments, each sentinel agent inde-

[1] An agent *id* is computed from its parent *id* $\times 4+$ an index (1-based if the parent is a commander, otherwise 0-based)

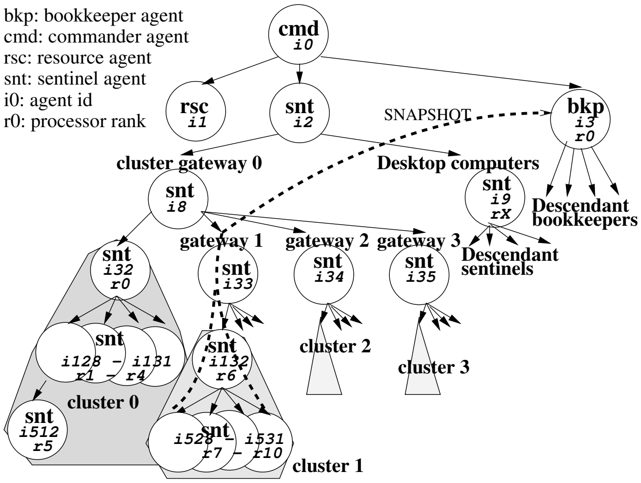

Figure 4. Inter-cluster job deployment

pendently migrates to its corresponding cluster node in accordance with the following procedures: **(1) find its gateway agent id** by dividing the given agent *id* by 4 as storing each quotient in an array till it reaches 8, and thereafter by checking the array elements in a reverse order from bottom to top till encountering the first 4-divisible non-8 element whose right quotient is the gateway agent *id*; **(2) index the given gateway's IP name in the gateway list** by repeatedly dividing its gateway agent *id* by 4 as storing each remainder in an array till the quotient reaches 8, and thereafter by checking each $remainder[i]$ in a reverse order as calculating $index = index \times 3 + remainder[i]$ where $index$ is first initialized to 0; and **(3) index its cluster node's IP name in the node list below the given gateway** by repeatedly dividing the agent *id* by 4 till the quotient reaches the gateway's leftmost children, and subsequently performing multiplicative computation: $index = index \times 4 + remainder[i] + 1$.

In Figure 4, sentinel 531's quotients by 4 include [132, 33, 8]; 132 is the first 4-divisible non-8 element; and its right quotient is 33. This is sentinel 531's gateway agent. Then, gateway agent 33 generates 8 and 1 as the quotient and remainder respectively when dividing its *id* by 4. Therefore, its index is: $0 \times 3 + 1 = 1$, which corresponds to cluster 1. Sentinel 531 generates 132 and 3 as the quotient and remainder respectively when dividing its *id* by 4. Therefore, its index in cluster 1 is: $0 \times 4 + 3 + 1 = 4$.

3.3 Job Monitoring and Check-Pointing

Each agent monitors the aliveness of its parent and children by receiving and sending a periodical ping message respectively. Due to the nature of our two-level hierarchy (shown in Figure 4), such ping messages are passed within either only the public address or a cluster-private domain but not across different private domains.

The UWAgents execution platform has been enhanced to cache a message receiver's IP address in the sender agent upon receiving the receiver's acknowledgment. This new feature allows a sentinel agent to deliver repetitive snapshots straightly to its bookkeeper, which is particularly effective when a gateway agent sends huge cluster-internal snapshots to its corresponding bookkeeper without interfering other gateways. For instance, sentinel 33 can collect all snapshots from sentinel 32 and 528 through to 531, and then deliver them directly to bookkeeper 3 or its descendant.

Another new feature is that two or more bookkeeper agents can reside on the same host as well as receive and distinguish snapshots from multiple sentinels. This saves the number of both computing nodes and bookkeeper agents to maintain execution snapshots.

3.4 Job Resumption

Figure 5 describes AgentTeamwork's inter-cluster job resumption. Upon an initialization, each sentinel agent receives additional arguments including a list of extra cluster nodes and a list of extra clusters.

The former list is generated by reserving some nodes of each cluster for future job resumption. As shown in Figure 5's case A, a crashed sentinel is resumed at a reserved node within the same cluster by its parent or child.

The latter list is used to resume a crashed gateway agent at a different cluster as shown in Figure 5's case B. The crashed gateway agent must be actually taken care of by its parent or one of its non-leftmost children, (i.e., another gateway agent). Resumed at one of extra clusters, the gateway agent is then supposed to spawn all descendants within this new cluster. All the previous children of this crashed gateway receive no more ping messages, realize that they all become network unreachable, and therefore send a suicide message to themselves as well as their descendants.

The final work of job resumption is to recover all broken inter-process TCP connections. Each gateway agent automatically instantiates a user program wrapper that simply starts GridTcp without launching any user program. GridTcp monitors and captures inter-cluster TCP messages in its gateway agent's snapshot. Therefore, upon a gateway resumption, GridTcp can recover all broken inter-cluster connections from the latest snapshot and retrieve all lost messages that have been in transit through the gateway.

4 Performance Evaluation

We have evaluated the system performance for job deployment, termination, and resumption over two cluster systems, named *cluster-r* and *cluster-i*. As summarized below, both clusters include 32 computing nodes, while *cluster-r* is faster than cluster-i.

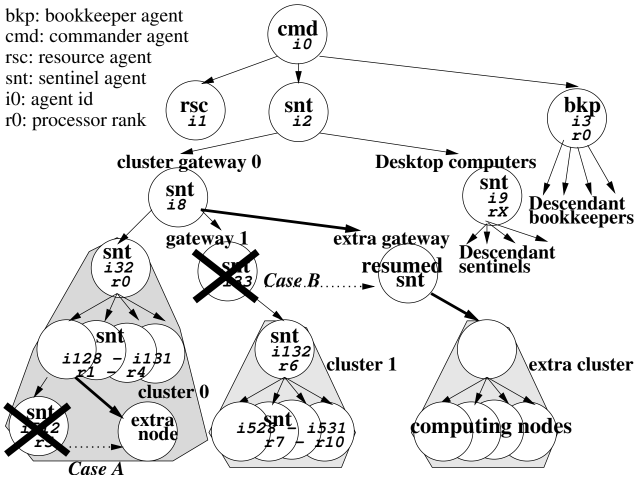

Figure 5. Inter-cluster job resumption

Cluster-r: 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

Cluster-i: 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

4.1 Deployment Performance

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

Figure 6 compares their performance. For the *depth first* scenario, AgentTeamwork always performed faster than Globus/OpenPBS, however both systems increased their job-deployment overhead sharply from 48 to 64 computing nodes. This is because the test program made a half of worker processes communicate with the master on the other cluster. For *breath first*, Globus/OpenPBS fluctuated its performance till 32 nodes while increasing more eminent overheads due to its linear job deployment over multiple clusters. On the other hand, AgentTeamwork showed its logarithmic increase of job-deployment overhead.

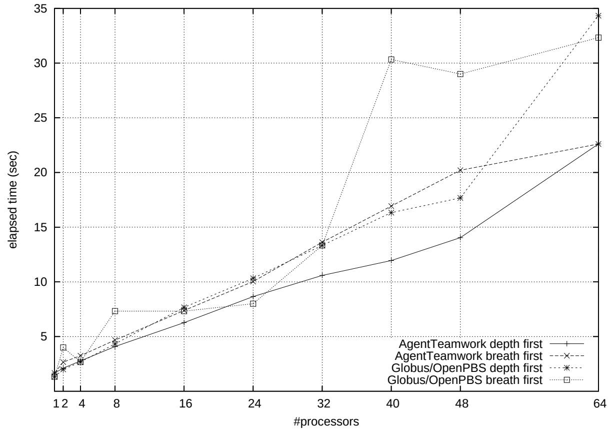

Figure 6. Job deployment effect by cluster node allocation.

4.2 Check-Pointing Performance

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

As shown in Figure 7, the non IP-caching deployment shows a super-linear increase of its overhead. There are two reasons. One is that all snapshots had to be funneled through the commander agent. The other is that each gateway itself must take periodical snapshots including all *GridTcp* messages passing through it, which burdens the gateway with relaying not only its own but also the descendant gateways' snapshots. On the other hand, the IP-caching deployment with one bookkeeper has drastically improved its job deployment performance by allowing execution snapshots to be delivered directly to a bookkeeper. Furthermore, the IP-caching deployment with two book-

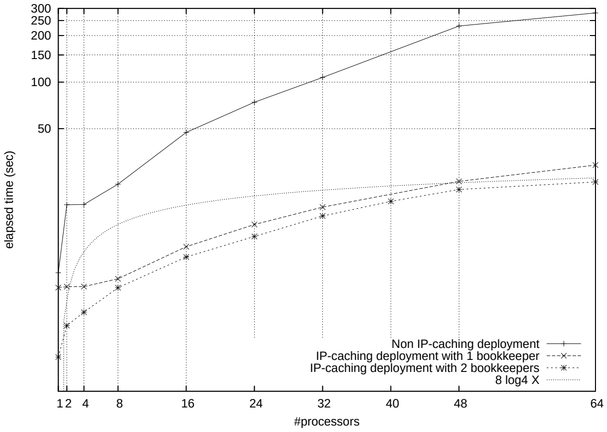

Figure 7. Job deployment effect by snapshot maintenance

keepers has balanced snapshot maintenance between their hosts to demonstrate the best performance mostly bounded by $log_4 N$.

5 Related Work

This section compares AgentTeamwork with other grid-computing systems in terms of (1) hierarchical job deployment and scheduling and (2) programming support for multi-cluster computing and fault tolerance.

5.1 Hierarchical Job Deployment

Job scheduling within a single domain or a cluster system has been facilitated in the form of both research prototypes and commercial products such as GRAM and PBS [5, 13]. It is a natural upgrade to orchestrate them with a centralized co-scheduler.

Nimrod-G targets multiple clusters by continuously submitting a parameter-sweep job to a remote GRAM running at a light-loaded domain [1], which does not however mean that Nimrod-G co-allocates multiple clusters to a single job.

DUROC is a Globus-based co-allocator that receives an RSL-described job submission from a user, dispatches the job to GRAMs (each running at a different cluster), collects a job status from them, and informs the user of a job suspension and termination [6]. However, the actual resource selection is left to a higher-level resource broker such as DCS (Dynamic Co-Allocation Service) [17].

Similarly, Condor-G uses a central co-allocator termed *GridManager* to deploy a job to remote GRAMs [8]. Condor flocking prepares in each domain at least one gateway machine to exchange a list of available machines with other domains [7]. It can mitigate a potential performance bottleneck incurred in a central manager, and differentiate domain-local users from external users.

DIET uses a tree of scheduler agents where a job request is forwarded to servers located at leaves of the tree and best fitted to the job [3]. The main difference from AgentTeamwork is that DIET's agents are location-static and dedicated to resource maintenance as well as scheduling but not to job monitoring and resumption.

5.2 Programming Support for Multi-Cluster Computing and Fault Tolerance

MPICH-G2, (i.e., the Globus version of MPICH) introduces two network topological parameters such as depths and colors, with which computing nodes can be grouped in a different communicator with the same depth or color [15]. The actual topological description must be defined in RSL, for which purpose the *mpirun* command automatically converts a user's machine file in an RSL file that is then passed to DUROC. Needless to say, fault recovery must be supported at lower platforms than at MPICH-G2.

Legion provides users with a fault tolerant version of MPI named *MPI-FT* that repeatedly takes a consistent snapshot of a job running on all computing nodes [12]. Similar to AgentTeamwork's user-initiated check-pointing operations, applications in Legion need to explicitly save their own data with *MPI_FT_Save*, to check their resumption with *MPI_FT_Init*, and to retrieve the latest snapshot from *MPI_FT_Restore*. The difference from AgentTeamwork is that their snapshots are maintained by a single file server and in-transit file contents must be recovered at a user level.

Condor-MW enhances fault tolerance of PVM applications by allowing them to react to runtime exceptions such as a suspension, a resumption, and a deletion of a remote host through *pvm_notify* and *pvm_recv* [4]. Its restriction, however, is the master-worker programming model in that only the master process can handle these events and take snapshots including communication with slaves (but not inter-slave communication).

6 Conclusions

We have applied mobile agents for fault-tolerant distributed job execution over multi-clusters that are not systematically managed by their local schedulers and turned on/off independently. The paper has proposed a two-level hierarchical agent-based job coordination including job deployment, monitoring, check-pointing, and resumption as well as inter-cluster MPI rank allocation and TCP connection establishment. Our performance evaluation demonstrated AgentTeamwork's job deployment in a logarithmic order, the effectiveness of direct snapshot transfer to and distributed maintenance at bookkeeper agents, and the importance of cluster node allocation in a depth first strategy.

Our next work is to develop a runtime job co-scheduling algorithm in addition to the job deployment infrastructure focused on by this paper. With this enhancement, we feel that mobile agents will be more practicable for fault-tolerant job execution over multiple clusters.

Acknowledgments

We are very grateful to Mr. David Grimmer, a UWB Linux systems administrator for all his technical support in clusters configuration and maintenance.

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/G: Killer application for the global grid? In *Proc. of the 14th International Symposium on Parallel and Distributed Processing – ISPDP*, pages 520–528, Cancun, Mexico, May 2000. IEEE-CS.

[2] A. I. D. Bucur and D. H. J. Epema. The performance of processor co-allocation in multicluster systems. In *Proc. of the 3rd IEEE/ACM Internationl Symposium on Cluster Computing and the Grid – CCGrid2003*, pages 302–309, Tokyo, Japan, May 2003.

[3] Pushpinder Kaur Chourhan, Holly Dail, Eddy Caron, and Frederic Vivien. Automatic middleware deployment planning on clusters. *International Journal of High Performance Computing Applications*, Vol.20(No.4):517–530, Winter 2006.

[4] Condor MW Homepage. http://www.cs.wisc.edu/condor/mw/.

[5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, Orland, FL, March 1998. IEEE-CS.

[6] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *Proc. of the 8th IEEE Symposium on High Performance Distributed Computing – HPDC8*, pages 219–228, Redondo Beach, CA, August 1999.

[7] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: load sharing among workstation clusters. Technical Report DUT-TWI-95-130, Delft University of Technology, The Netherlands, 1995.

[8] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proc. of the 10th IEEE Symposium on High Performance Distributed Computing – HPDC10*, pages 55–63, San Francisco, CA, August 2001.

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

[10] 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.

[11] 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.

[12] Integrating Fault-Tolerance Techniques in Grid Applications. *Anh Nguyen-Tuong*. PhD thesis, University of Virginia, Charlottesville, VA 22904, August 2000.

[13] James Patton Jones. PBS Pro Relase 5.1. User guide, Veridian Systems, Inc., Mountain View, CA, November 2001.

[14] William M. Jones, Luis W. Pang, Dan Stanzione, and Walter B. Ligon III. Bandwidth-aware co-allocating meta-schedulers for mini-grid architectures. In *Proc. of the IEEE International Conference on Cluster Computing – Cluster2004*, pages 45–54, San Diego, CA, September 2004. IEEE-CS.

[15] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, Vol.63(No.5):551–563, 2003.

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

[17] J. M. P. Sinaga, H. H. Mohamed, and D. H. J. Epema. A dynamic co-allocation service in multicluster systems. In *Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 194–209, New York, NY, June 2004. LNCS 3277.