Resource Management and Monitoring in AgentTeamwork Grid Computing Middleware

Munehiro Fukuda, Cuong Ngo, Enoch Mak, and Jun Morisaki
Computing and Software Systems, University of Washington, Bothell
18115 Campus Way NE, Bothell, WA 98033
Email: mfukuda@u.washington.edu

***Abstract*— The successful key to resource management in grid computing would include language design for heterogeneous resource description, database design for dynamic resource repository, and tool design for scalable resource monitoring. Particularly focusing on simultaneous use of multiple clusters for grid computing, we are currently addressing these problems by implementing an XML-based resource database and resource-monitoring mobile agents in the AgentTeamwork grid-computing middleware system. The system describes each cluster in an XML file that even groups cluster nodes based on their resource type. The database returns a collection of clusters or sub cluster groups to a job. The system also deploys a hierarchy of mobile agents to remote nodes for parallel resource monitoring. This paper presents our implementation techniques and preliminary performance of resource management and monitoring in AgentTeamwork.**

I. Introduction

Most challenges of resource management in grid computing would originate from the large number, heterogeneity, and dynamic availability of remote computing resources as well as their network connectivity. The design issues include language for describing heterogeneous resources, database for registering and retrieving dynamic resource information, and tools for monitoring numerous resources.

Consider a scientific-computing job that needs as many computing nodes as possible. It is always better to allocate to the job a cluster or even multiple cluster systems than a collection of heterogeneous desktop machines. For language design, Globus allows the job to specify particular cluster systems in an RSL-based description of its resource requirements [1]. For database design, Condor describes both remote resources and job requirements in ClassAd, registers resources in a central matchmaker, and performs gang-matching to allocate to the job the best fitted cluster of computing nodes simultaneously [2]. Similarly, Legion collects status of remote resources into a central information database named Collection [3]. For resource monitoring, NWS (Network Weather Service) facilitates a collection of libraries that capture dynamic resource and network status at remote sites [4]. Condor can automatically launch NWS at each remote site that periodically sends back its up-to-date ClassAd to a central matchmaker.

Despite such pioneer work, some challenges still remain in resource management particularly of multiple cluster systems. The first problem is that a cluster is not always composed of homogeneous nodes and/or network. This implies that we may need to divide and to describe a cluster into multiple logical groups. This problem also necessitates a mechanism to allocate to a job a collection of sub cluster groups rather than clusters themselves. The second problem is that one-by-one status queries to all remote resources become a performance bottleneck. This in turn means the necessity of developing a parallel resource-monitoring/collecting mechanism.

We are currently addressing these two problems by implementing an XML-based resource database and resource-monitoring mobile agents in the AgentTeamwork grid-computing middleware system [5]. The system describes each cluster in an XML file that even groups cluster nodes based on their resource type. The database returns a collection of clusters or sub cluster groups to a job. The system also deploys a hierarchy of mobile agents to remote nodes for parallel resource monitoring.

This paper discusses particularly about resource management and monitoring in AgentTeamwork rather than its job deployment we previously described in [5]. The rest of the paper is organized as follows: section 2 gives a system overview of AgentTeamwork; section 3 focuses on resource management; section 4 explains our resource-monitoring algorithm; section 5 previews AgentTeamwork's monitoring performance; and section 6 concludes our discussions.

II. System Overview

AgentTeamwork is a grid-computing middleware system that coordinates parallel and fault-tolerant job execution with mobile agents [5]. A new computing node can join the system by running a UWAgents mobile-agent execution daemon to exchange agents with others [6]. The system distinguishes several types of agents such as commander, resource, sensor, sentinel, and bookkeeper agents, each specialized in job submission, resource selection, resource monitoring, job deployment, and execution bookkeeping respectively.

A user submits a new job with a commander agent that passes the job's resource requirements to and then receives a collection of remote machine names from a resource agent. Spawned by the resource agent, sensor agents keep monitoring remote computing nodes in their hierarchy. The commander spawns a pair of sentinel and bookkeeper agents, each hierarchically deploying as many children as the number of the remote machines. A sentinel launches a user process at a different machine with a unique MPI rank, takes an

execution snapshot periodically, sends it to the corresponding bookkeeper, checks the activity of its parent and child agents, and resumes them upon a crash. A bookkeeper 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 and check-pointed by a user program wrapper, one of the threads running within a sentinel agent. The wrapper internally facilitates error-recoverable TCP and file libraries, each named *GridTcp* and *GridFile* respectively, which serialize and de-serialize an execution snapshot with in-transit and even inter-cluster messages as well as buffered file contents. A user program can take advantage of these fault-tolerant features by inheriting the *AteamProg* class that has re-implemented Java socket, file, and even mpiJava classes [7] with *GridTcp* and *GridFile*. An execution snapshot is taken manually at the user level by calling *AteamProg.takeSnapshot( )* or automatically at the system level upon each transition from one to another system-framed function that contains a portion of application code.

In the following, we will focus on how a resource agent interacts with an XML database for resource registration and retrieval as well as collects dynamic status of remote resources through a hierarchy of sensor agents.

III. Resource Management

AgentTeamwork uses XML to describe computing resources because of not only its flexibility of content definition but also its availability of language processing and database management tools. Each XML file in AgentTeamwork can describe resources either of an independent desktop machine or a cluster system. Although such an XML resource file uses trivial tags regarding CPU, memory, disk, and network, it distinguishes the following three tags to handle the runtime resource status of a cluster system that may consist of heterogeneous computing nodes:

1) **design_time** describes the factory-initialized resource specification of a desktop or a cluster.
2) **run_time** maintains the runtime resource status that is periodically updated by the system.
3) **group** may appear one or more times both in the design_time and run_time tags to classify cluster nodes into logical groups based on their resource specification.

Figure 1 gives an XML example to describe the resources of the *medusa* cluster addressed with *medusa.uwb.edu* and *medusa-myr* in the public and its private domain respectively. This cluster consists of two processor groups, each characterized with its different CPU speed and network bandwidth, (i.e., 2.8GHz/2Mbps and 3.2GHz/1Mbps respectively).

AgentTeamwork is based on a two-level XML repository where users first register their XML resource files in a shared ftp server, (e.g., *ftp.tripod.com*), and thereafter each user's resource agent downloads only new files into its local XML database just once a day. The actual XML maintenance, retrieval, and update take place locally at each user's XML database. As planning on future customization for grid computing, we have implemented our own XML database that includes the following components:

```
1   -<cluster>
2    -<design_time>
3      <domain>UWB</domain>
4      <name>medusa</name>
5      <gateway>medusa.uwb.edu</gateway>
6      <alias>medusa-myr</alias>
7     -<group>                          <!-- sub group #1 -->
8      -<ip_list>        <!-- cluster-private ip names -->
9        <ip_name>mnode0</ip_name>
10       ...
11       <ip_name>mnode7</ip_name>
12     </ip_list>
13     <cpu_speed>2800</cpu_speed>           <!-- MHz -->
14     <cpu_arch>i386</cpu_arch>
15     <cpu_count>1</cpu_count>     <!-- #cpus per ip -->
16     <memory>512</memory>                   <!-- MB -->
17     <disk_space>60</disk_space>            <!-- GB -->
18     <intra_net_band>2000</intra_net_band><!-- Mbps -->
19    </group>
20   -<group>                          <!-- sub group #2 -->
21    -<ip_list>        <!-- cluster-private ip names -->
22      <ip_name>mnode8</ip_name>
23      ...
24      <ip_name>mnode31</ip_name>
25    </ip_list>
26    <cpu_speed>3200</cpu_speed>           <!-- MHz -->
27    <cpu_arch>i386</cpu_arch>
28    <cpu_count>1</cpu_count>     <!-- #cpus per ip -->
29    <memory>512</memory>                   <!-- MB -->
30    <disk_space>36</disk_space>            <!-- GB -->
31    <intra_net_band>1000</intra_net_band><!-- Mbps -->
32   </group>
33  </design_time>
34  -<run_time>
35    ...                     <!-- filled at run time -->
36   </run_time>
37  </cluster>
```

Fig. 1. An XML-based resource description

1) **XDBase.java** is our database manager that maintains two collections of XML-described resource files in a DOM format: one maintains initial XML files downloaded from a shared ftp server and the other is specialized to XML files updated at run time for their latest resource information. Upon a boot, *XDBase.java* waits for new service requests to arrive through a socket.
2) **Service.java** facilitates basic service interface to *XDBase.java* in terms of database management, query, deletion, retrieval, and storage, each actually implemented in a different sub class. *Service.java* receives a specific request from a user either through a graphics user interface or a resource agent, and passes the request to *XDBase.java* through a socket.
3) **XDBaseGUI.java** is an applet-based graphics user interface that passes user requests to *XDBase.java* through *Service.java*.
4) **XCollection.java** is a collection of sub classes derived from *Service.java*, each implementing a different service interface. A resource agent carries *XCollection.java* with it for the purpose of accessing its local *XDBase.java*.

Among these components, the *XDBase* manager serves as a matchmaker between resource requirements received from a user (thus through his/her commander and resource agents)

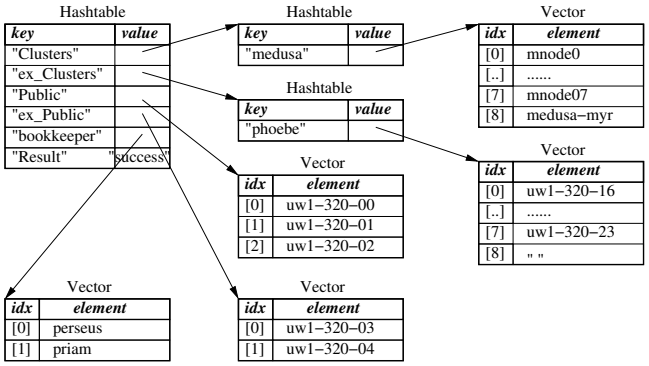

Fig. 2. A response to an Xpath resource request

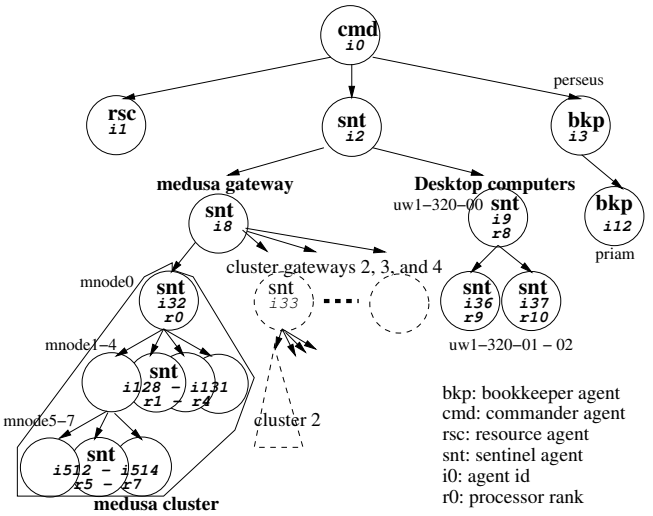

Fig. 3. Hierarchical job deployment

and XML resource files retrieved from its local collections.

Resource requirements distinguish the resource types corresponding to the XML tags defined in Figure 1 as well as the following five additional parameters including *total*: the total number of computing nodes, *multiplier*: a reservation for additional nodes to be used for future job migration, *bookkeepers*: the number of bookkeepers to save execution snapshots, *cluster*: an address of a specific cluster, and *ip*: addresses of specific desktop computers. They are passed from a resource agent to its local *XDBase* in three *Xpath* statements such as *cluster*, *runtime*, and *public*, each respectively requesting a specific cluster, candidate clusters, and candidate desktops fitted to a given job.

XML resource files are retrieved in response to those *Xpath* statements and converted by *XDBase* into a hash table as illustrated in Figure 2 (named an itinerary in the following discussions). The *"Clusters"* and *"ex_Clusters"* keys point to clusters to be allocated to a job and those to be reserved for future job migration respectively. Similarly, the *"Public"* and *"ex_Public"* keys refer to desktops allocated for a job and reserved for job migration. The *"bookkeeper"* key specifies a collection of desktops to accept a bookkeeper.

Given this itinerary back from the local *XDBase*, a commander agent deploys sentinel and bookkeeper agents in a hierarchy as shown in Figure 3. In particular, sentinel agents form two subtrees such as the left and right dedicated to clusters and independent desktops respectively. For the left subtree, the leftmost sentinel at each layer and its descendants are deployed to computing nodes in the same cluster while the other agents migrate to a different cluster gateway. This deployment makes it easier to manage all cluster gateways in an outer hierarchy, (i.e., a tree that trims off its leftmost subtree at each layer) and all computing nodes in an inner hierarchy, (i.e., the leftmost subtree), which localizes interprocess communication within each cluster.

IV. Resource Monitoring

A resource agent takes charge of not only planning on a new itinerary for a job but also deploying sensor agents that keep reporting remote resource status to the local *XDBase*. We must first choose only one of multiple resource agents that are engaged in a different job execution but working with the same *XDBase*. The selection is based on how frequently they are instructed by their user to probe remote resources. If a new resource agent has a higher probing frequency than the current value registered in *XDBase*, it becomes the primary agent that will deploy sensor agents in a hierarchy, whereas the previous primary will terminate all its sensor agents simultaneously.

A pair of sensor agents, each deployed to a different node, periodically measure the usage of CPU, memory, and disk space specific to their node as well as their peer-to-peer network bandwidth, (using *uptime*, *free*, *df*, and *ttcp* commands respectively). These two agents are distinguished as a client and a server sensor. The client initiates and the server responds to a bandwidth measurement. They spawn child clients and servers respectively, each further dispatched to a different node and forming a pair with its correspondence so as to monitor their communication and local resources.

As shown in Figure 4, the sensor agents' inter-cluster deployment is similar to that of sentinel agents, while sensors must form pairs of a client and a server. The primary resource agent spawns two pairs of root client and server sensors, one dedicated to desktop computers and the other deployed to clusters. The former pair recursively creates child clients and servers at different desktops in the public network domain. The latter pair migrate to different cluster gateways where they creates up to four children. Two of them migrate beyond the gateway to cluster nodes as further creating offspring. The other two are deployed to different cluster gateways, and subsequently repeat dispatching offspring to their cluster nodes and other cluster gateways. With this hierarchy, the resource information of all cluster nodes is gathered at their gateway and thereafter relayed up to the resource agent that eventually reflects the latest status to the local *XDBase*.

V. Monitoring Performance

We have compared AgentTeamwork with NWS for their monitoring accuracy, using a Giga-Ethernet cluster of 24 Dell

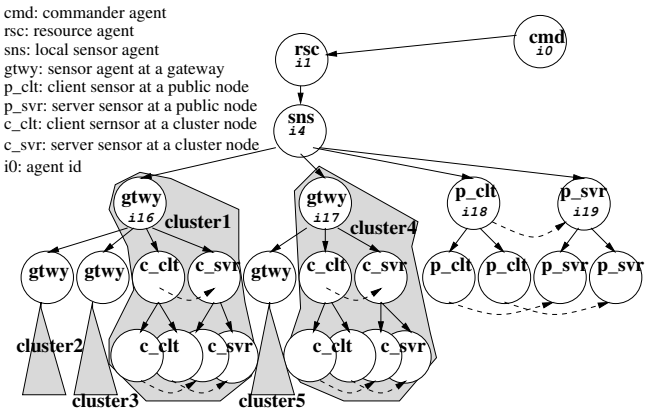

Fig. 4. Resource monitoring in an agent hierarchy

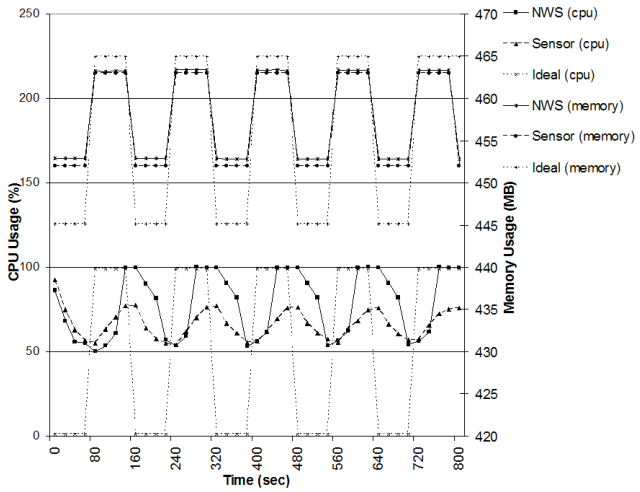

Fig. 5. CPU load monitored by AgentTeamwork and NWS

computing nodes, each with 3.2GHz Xeon, 512MB memory and 36GB SCSI hard disk. During our evaluation, we have executed an mpiJava test program that repeats wasting CPU power, memory space, disk space, and network bandwidth at a different node respectively for every 80-second interval.

Figure 5 shows CPU load and memory usage monitored by AgentTeamwork and NWS when the mpiJava program has repeatedly executed Tower of Hanoi and allocated a 20M-byte array respectively. For CPU load, AgentTeamwork was slow to react to variable load due to its use of uptime. In contrast, NWS performed better by choosing the most accurate value from uptime, vmstat, and its own estimation algorithm. For memory usage, both systems demonstrated the same accuracy. Note that they have estimated only 10M-byte memory deallocation due to JVM's delayed garbage collection.

For network bandwidth and disk space, (although there is no space to show the results in a graph), our evaluation has revealed that NWS can measure 4M-byte network transfers better than AgentTeamwork, whereas both systems have accurately reacted to repetitive 50M-byte disk allocation. The main reason is that NWS uses an neural-network algorithm named Network Bandwidth Predictor.

No.	Monitoring tasks	Maximum time (seconds)
A	CPU, memory, and disk	$0.001 \sim 0.02$
B	Network bandwidth	~ 2.0
C	A report from a server to a client agent	~ 3.0
D	A report from a client to a parent agent	$0.01 \sim 3.0$
	Maximum total of local tasks: A + B + C	5.02

TABLE I
TIME REQUIRED FOR ONE MONITORING CYCLE

We have also measured the time required for one cycle of resource monitoring at each computing node. As summarized in Table I, each pair of client and server sensor agents take 5.02 seconds to complete their local monitoring task. Each client sensor needs maximum 3.0 seconds to report results to its parent. While the monitoring task can be carried out in parallel among all pairs, each reporting task must be serialized at the same parent. Since two pairs of agents report to the same parent, one monitoring cycle takes at most $5.02 + 3.0 \times 2 = 11.02$ seconds. Given N computing nodes, we have $N/2$ pairs of server and client agents whose entire monitoring latency is estimated as $11.02 log_2 N/2 = 11.02(log_2 N - 1)$ seconds.

VI. CONCLUSION

Focusing on job deployment over multiple clusters, we have presented AgentTeamwork's resource management and monitoring, each implemented in an XML-based database manager and in a hierarchy of mobile agents. We are planning on two tasks: (1) to have the database prioritize resources in their current availability and previous usage, and (2) to include NWS in sensor agents for more accurate resource monitoring.

ACKNOWLEDGMENT

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

REFERENCES

[1] K. Czajkowski, I. Foster, and C. Kesselman, "Resource co-allocation in computational grids," in *Proc. of the 8th IEEE Symposium on High Performance Distributed Computing – HPDC8*, Redondo Beach, CA, August 1999, pp. 219–228.

[2] R. Raman, M. Livny, and M. Solomon, "Policy driven heterogeneous resource co-allocation with gangmatching," in *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing – HPDC-12*, Seattle, WA, June 2003, pp. 80–89.

[3] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw, "The Legion resource management system," in *Proc. of the 5th Workshop on Job Scheduling Strategies for Parallel Processing – JSPP'99*. San Juan, Puerto Rico: Springer Verlag, April 1999, pp. 162–178.

[4] R. Wolski, "Experiences with predicting resource performance on-line in computational grid settings," *ACM SIGMETRICS Performance Evaluation Review*, vol. Vol.30, no. No.4, pp. 41–49, March 2003.

[5] M. Fukuda, K. Kashiwagi, and S. Kobayashi, "AgentTeamwork: Coordinating grid-computing jobs with mobile agents," *International Journal of Applied Intelligence*, vol. Vol.25, no. No.2, pp. 181–198, October 2006.

[6] M. Fukuda and D. Smith, "UWAgents: A mobile agent system optimized for grid computing," in *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*. Las Vegas, NV: CSREA, June 2006, pp. 107–113.

[7] mpiJava Home Page, "http://www.hpjava.org/mpijava.html."