Inter-Cluster Job Coordination Using Mobile Agents

Munehiro Fukuda [1] Yuichiro Tanaka [1] Luis Miguel Campos [2] Shinya Kobayashi [3]

[1] Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan
fukuda@is.tsukuba.ac.jp, yuichiro@padc.mmpci.is.tsukuba.ac.jp

[2] Information and Computer Science, University of California, Irvine, CA 92697, USA
lcampos@ics.uci.edu

[3] Department of Computer Science, Ehime University, Matsuyama, Ehime 790, Japan
kob@cs.ehime-u.ac.jp

Abstract

Cluster computing is becoming increasingly popular among users of parallel and distributed applications. However, since few clusters are solely dedicated to run individual user jobs, it is necessary to coordinate those jobs among independently administered clusters effectively and with less user interaction. For this purpose, we have developed an infrastructure for inter-cluster job coordination using Voyager mobile agents. Once a user submits a job request, which is converted into XML, a mobile agent searches for the most available cluster, establishes a job execution environment, executes the job, and reports its results back to the client. Using Voyager's dynamic aggregation feature, we also plan to make such job coordination agents to evolve themselves as to be suitable to be deployed in dynamic cluster environments. This paper presents the design principles and the latest implementation status of our infrastructure as well as advanced services using evolvable agents.

1 Introduction

The emergence of commodity high-performance microprocessors and high-speed networks has made it attractive to use cluster systems as a commodity computing platform. Many novice users have joined the ranks of those using cluster computing systems, however they currently suffer from the burden of hectic job coordination. The main reason for the current situation is because few clusters are physically and/or temporally dedicated to each user's job. The responsibility to search for, reserve, upload their programs to, and download results from the most available cluster lies squarely on the users.

Such cluster job coordination is categorized in grid computing as discussed in [5]. However most existing grid-computing environments so far have been designed aiming for highly-skilled and internationally-active researchers in parallel programming. Some systems request users to assemble their job control scenario (such as via a Unix shell) using a variety of grid-computing commands; others to embed resource-accessing functions in their application program. Furthermore, most systems need to construct a centralized directory service shared by all participating computing nodes. Such prerequisites present a significant hurdle to novice users who simply wish to dispatch their jobs to the most available cluster within their closed computing community.

Therefore, our interest is in relieving users from the task of inter-cluster job coordination rather than providing them with precise resource-accessing options. With this in mind, we have applied the mobile agent paradigm to achieve job coordination. Upon receiving a user's job request, an agent acting on behalf of the user navigates over the network in search for a cluster satisfying the request, coordinates the job execution, and reports computation results back to the user. The autonomy of mobile agents not only allows for the offloading of the job coordination task from the user but also has the potential to simplify the maintenance and upgrading of remote clusters. We also provide mobile agents with evolvability, that is to incorporate new executable objects into them at each site they visit and make their behavior adaptable to the dynamic network environment, which thus keep mobile agent code simple and maintainable.

Currently, we are implementing an inter-cluster job coordination infrastructure using Voyager mobile agents [12], XML, servlets [10] and FastCGI [13]. In this paper, we discuss problems in using remote clusters in Section 2, give the design principles in Section 3, explain the latest implementation status in Section 4, introduce the extended services provided by our infrastructure in Section 5, and finally present our conclusions and future work in Section 6.

2 Background

The authors and their students have had the opportunity to use three distinct PC clusters at University of Tsukuba, an NFS connection of SPARC workstations at UC Irvine, and an AP3000 server at Ehime University. From the experience gained by using these clusters, we have learnt the following three lessons:

1. *Cluster search:* Users must repeat *rlogin/rsh/ssh* to each computing node of a cluster or even to a different cluster until finding a necessary number of idle nodes. This exhaustive search cannot be always automated in a simple shell program because of the reason that some server must be accessed along a predefined route requiring step-by-step *rlogin*s.

2. *Job coordination:* Especially when using a remote cluster, users must export code, recompile it if necessary, modify a configuration file (like a MPI host file for example), and import results before and after computation. This series of job coordination tasks varies depending on the configuration and availability of each cluster, and thus can not easily be arranged in automated shell scripts.

3. *Environment customization:* Some clusters may be protected from others by a firewall or invisible due to their private IP address. This discourages users to use multiple clusters simultaneously in order to scale up the parallelism of application programs, unless the user prepares proxies or uses HTTP tunneling to permit communication among those clusters at least during the duration of job execution.

Several possible solutions that address the problems we have described have already been deployed.

The Globus toolkit [4] consists of a collection of basic grid-computing services. For instance, MDS serves as the system-wide directory, with which we could perform cluster search. Nexus provides a uniform interface between high-level parallel programs such as MPI and HPC++ and low-level communication layer such as IP and ATM. However, distributed job coordination still lies on each user, who must precisely describe all coordination procedures in a resource specification language.

Ninf [11] and Netsolve [2] are client-server oriented environments. They dynamically locate the servers for, flexibly marshal arguments of, and schedule the sequential/parallel invocation of RPCs. Those systems intend to orchestrate each application's RPCs over the network or coordinate its access to special-purpose servers. This in turn implies that they are not suitable to search for and export an entire job to the most available cluster.

Legion [7] is an object-oriented environment that regards all hardware and software resources as uniform objects, thus encapsulating their details, and realizing distributed transparency of those objects. The system coordinates a user job using three components termed, Collection, Scheduler, and Enactor, responsible respectively for collecting resource state information, mapping user requests to resources, and activating those resources for a user job. However, Legion assumes that network resources and protocols in use will not change.

Catalina [8] and an active grid-monitoring system using MAP agents [16] are innovative grid-computing environments using mobile agent technology. In Catalina, an application delegated manager (ADM), assigned to each application, launches a mobile agent termed a "task agent" to monitor and manage each task execution of its application. Their agents are characterized as sentinels reporting their job status to and ordered a next behavior by ADM, the commander to centralize system status in it. In MAP, on the other hand, each agent has its different speciality such as collecting computation results and trapping system events, and therefore a cooperative work of independent agents perform an entire user application. However, grid resource information is still centralized into the Directory Service component.

In contrast with those systems, our pursuit of inter-cluster job coordination is to use cognitive and evolvable mobile agents, each acting on behalf of the user to coordinate the job execution as navigating over and changing their behavior adaptable to the decentralized and dynamic network environment.

Our proposed environment is composed of five main modules: (1) XML as a resource description language, (2) servlets as user and system administrator interface, (3) per-cluster agents as resource managers, (4) mobile agents as user representatives, and (5) FastCGI processes as inter-cluster communication proxies. Broadly speaking, each user orders a job request through a servlet which converts it into XML and deploys a mobile agent with this XML description. The agent negotiates with a per-cluster agent every time it visits a new cluster, launches a user job, and reports results to the client. The following two sections explain the structure and coordination sequence of our system.

3 Design Principles

Figure 1 illustrates the environment present in each cluster participating in our inter-cluster job coordination. (Non-shaded components are those being designed by us, while shaded ones are freeware.) We chose Voyager ORB [12] as the underlying mobile agent workbench. The main reason behind our choice was its dynamic aggregation feature permitting mobile agents to dynamically load new class objects

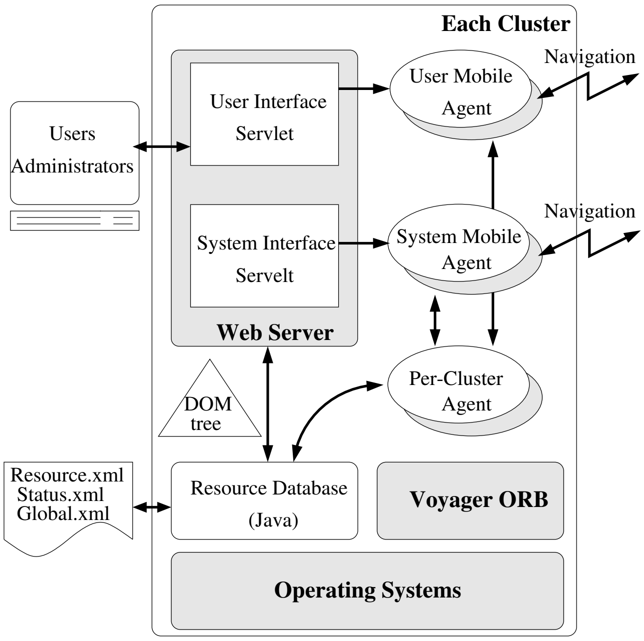

Figure 1. Job coordination infrastructure

into themselves and thus used to evolve agent behavior in response to runtime system reconfiguration. On top of Voyager ORB is a per-cluster agent that schedules a user job requested from his/her mobile agent. In addition, each cluster is supposed to run an HTTP server where two servlets are registered to provide interface to users and administrators respectively. The former submits a user mobile agent upon receiving a job request, while the latter may deploy a system mobile agent in case of a change of remote servers configuration. Those two types of mobile agents navigate over a predefined region of clusters (*is.tsukuba.ac.jp*, *ics.uci.edu*, and *cs.ehime-u.ac.jp* in our example) to complete their missions. The details of each component can be described as follows:

1. *Interface Servlets and Resource Database:* We employ XML to specify the infrastructure-wide status, each cluster's resource availability, and user job requests. Many XML document servers are composed of servlets and a database manager. The former provides front-end user interface, formats text input to, and generates HTML output from DOM trees, (i.e., XML-based syntax trees.) The latter converts DOM trees into XML files or vice versa. We apply this typical XML document technique for resource database. More specifically, a cluster administrator interacts with a system interface servlet which then forwards resource update information to the back-end XML cluster resource database. One of the advantages in using XML is the ease of DOM-tree generation and analysis. Therefore, the system interface servlet passes a DOM tree to a system mobile agent deployed to remote clusters in order to change their resource availability. Similarly, once receiving a job request, a user interface servlet passes the corresponding DOM tree to a user mobile agent as explained below.

2. *Per-Cluster Agent:* This is a Voyager agent local to each cluster and responsive to incoming user and system mobile agents, providing authentication and servicing their requests. Its service for a user mobile agent starts with receiving the job request in a DOM tree. The per-cluster agent then compares it with a DOM tree retrieved from its own XML cluster resource database, and reserves resources if this DOM matching in successful. The actual job execution must be initiated by each user mobile agent which may coordinate a job execution using two or more clusters. Another important job of the per-cluster agent is to provide user jobs with inter-cluster transparent communication especially over the firewall. For this purpose, we use an HTTP tunneling technique in which each per-cluster agent invokes a FastCGI process responsible for funneling all user communication to a remote HTTP server.

   The service for a system mobile agent is not only to update the local XML cluster resource database but also evolve the behavior of the per-cluster agent itself. Such runtime behavioral evolution is enabled by Voyager's dynamic aggregation in which a system mobile agent brings new code with it and a per-cluster agent incorporates the code into itself.

3. *User/System Mobile Agents:* They are active agents responsible for coordinating distributed computation and update system-wide resource information. A user mobile agent, upon submitted from a user interface servlet, navigates over the network, carrying a DOM tree with it. It negotiates with every per-cluster agent it visits until all user resource requests are satisfied through the DOM-tree matching mechanism previously described. Once the negotiation is completed, the user mobile agent generates a configuration file, (for example, an MPI host file) and launches a user process at each node/cluster it has reserved. Its final task is to bring back the job output to the client node.

   The system mobile agent carries with it not only DOM trees to update remote/global resource information but also Voyager-provided objects, termed *facets*, capable of containing new code to evolve remote per-cluster agents.

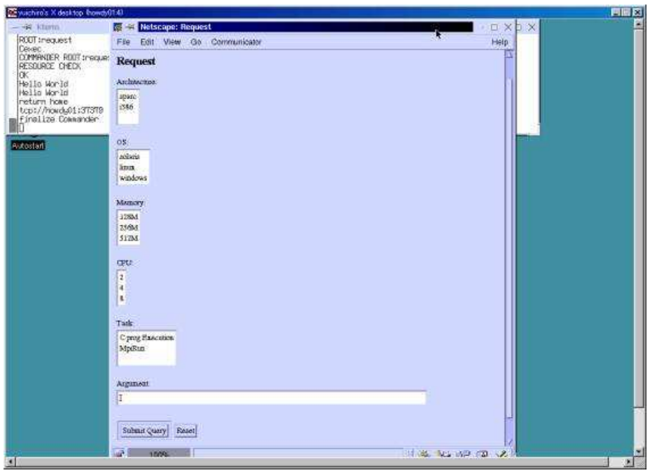

Figure 2. Resource specification and job submission

4 Implementation

Based on the design principles discussed above, we have constructed an inter-cluster job coordination infrastructure. In the following two sub-sections, we first describe the primary coordination features we have implemented so far, and thereafter introduce the TCP/*rsh* proxy we are designing to support transparent inter-cluster communication.

4.1 Job Coordination

A job coordination starts with an input of a user job request through the user interface servlet as shown in Figure 2. A user specifies the resource parameters necessary to accomplish his/her job: CPU architecture, operating system, memory space, the number of processors, the type of job such as MPIRUN, arguments passed to the job, and the job name.

Upon system initialization, the user interface servlet starts the Voyager ORB. It thereafter waits on its POST method for a user to submit a job request as an HTTP body. This method simply picks up all the resource parameters, instantiates a new DOM tree, and appends each parameter as a child of this DOM tree's root. (The actual DOM tree construction is straightforward as introduced in many XML textbooks [10].) Finally, it instantiates, initializes, and dispatches a user mobile agent.

Figure 3 shows the brief code of a user mobile agent, named *UserAg*. It receives a job name, a DOM tree, and a home URL in the *setJob* method (lines 7-10) and navigates over the network in *search* method (lines 11-15), both invoked by the user interface servlet. When arriving at the destination cluster, this agent resumes its execution from the *coordinate* method (line 16). It passes its DOM tree to the per-cluster agent's *negotiate* method to examine if the tree matches (a part of) the one maintained by the per-cluster agent (line 17). If that comparison results in success, the user mobile agent instructs the per-cluster agent to start its user job (line 18).

```
 (1) public class UserAg implements IUserAg,
 (2)                              Serializable {
 (3)  private String[] job;
 (4)  private Vector result = null;
 (5)  Document reqDOM;
 (6)
 (7)  public void setJob(String[] j, Document d,
 (8)                     String[] h) {
 (9)   job = j; reqDOM = d; home = h;
(10)  }
(11)  public void search() {
(12)   IClusterAg server
(13)    = (IPerClusterAg)Namespace.lookup(...);
(14)   Agent.of(this).moveTo(server,"coordinate");
(15)  }
(16)  public void coordinate(IClusterAg server){
(17)   if (server.negotiate(reqDOM)) {
(18)    server.exec(job);
(19)    result=server.getResult();
(20)   }
(21)   Agent.of(this).moveTo( home,"report");
(22)  }
(23)  public void report() {
(24)   for (Enumeration em = result.elements();
(25)        em.hasMoreElements();)
(26)    System.out.println(em.nextElement());
(27) }}
```

Figure 3. User Mobile Agent

Figure 4 gives the overview of a per-cluster agent, (i.e., *ClusterAg*). It instantiates a *myResource* object which maintains the latest resource DOM tree updated continuously by the resource database (lines 6-8). The *myResource* object also takes charge of comparing its resource DOM tree with the one passed from a user mobile agent (lines 9-11), and reserves the requested resource, (more specifically marks the corresponding nodes/leaves in its resource DOM tree.)

Another important task of the per-cluster agent is to customize the environment for each user job, achieved by the *exec* method. In Figure 4, this method includes only a user process execution (lines 13-16) and the standard output redirection to a *BufferredReader* object, *d* (lines 17-18), the latter returned as *status* to the client user mobile agent (lines 19-21). However, the *exec* method could eventually take care of redirecting all file I/O from its local storage to the user mobile agent and communicate with the local FastCGI process to implement over-firewall user task communication using HTTP tunneling technology (Section 4.2 gives the details).

```
 (1) public class ClusterAg
 (2)                 implements IClusterAg {
 (3)  Resource myResource;
 (4)  private Vector status=null;
 (5)
 (6)  public ClusterAg {
 (7)   myResource = new Resource();
 (8)  }
 (9)  public boolean negotiate(Document reqDOM){
(10)   return resource.resource_check(reqDOM);
(11)  }
(12)  public void exec(String[] args){
(13)   Runtime runtime=Runtime.getRuntime();
(14)   String line=null;
(15)   status = new Vector();
(16)   Process process = runtime.exec(args);
(17)   InputStream in = process.getInputStream();
(18)   BufferedReader d=new BufferedReader(..in..);
(19)   while ((line = d.readLine()) != null)
(20)    status.addElement(line);
(21)  }
(22)  public Vector getResult() {
(23)   return status;
(27) }}
```

Figure 4. Per Cluster Agent

Finally, the per-cluster agent deallocates all reserved resource by deleting the corresponding marks from the resource DOM tree. In the future implementation, that would be achieved when the user mobile agent calls *getResult* to receive the execution status (lines 22-23). With this execution status, the user mobile agent returns to its home cluster and displays the results (lines 21-26 in Figure 3).

4.2 Inter-Cluster Communication

In order to avoid consuming public IP addresses, most clusters employ private IPs. Only their gateway node is made directly accessible to remote users, while the other nodes must be logged in from that gateway. Furthermore, such clusters are generally secured by the firewall. As far as user jobs can be executed inside a cluster, the private IPs and firewall would work as expected. However, when it comes to large-scale jobs requiring multiple clusters simultaneously, these technologies would be obstacles to inter-cluster communication based on TCP.

There are several techniques to address this problem. The IP masquerade permits nodes in a private network to get access to Internet resources [9]. Many proxy servers such as Delegate [3], and Squid [14] forwards TCP packets to a port permitted to pass through the firewall. The resource manager beyond the firewall (RMF) [15] is a Globus proxy to make its underlying Nexus communication established beyond the firewall, so that remote users can submit their MPI applications, (modified to use Nexus communication,)

local gateway property			destination cluster		
gateway ip	port	rshs	gateway ip	node ip	port
192.168.1.1	5001	0	111.222.333.444	192.168.1.1	5000
192.168.1.5	5002	0	111.222.333.444	192.168.1.2	5000
192.168.1.5	514	1	111.222.333.444	192.168.1.3	514
192.168.1.5	5001	0	111.222.333.444	192.168.1.4	5000

Table 1. An example of translation table

to a firewall-protected cluster.

Yet, these techniques have not removed all the obstacles. In the IP masquerade, once a private node accesses a public Internet node, the gateway assigns a new port to forward responses from the public to the private node. This means that communication cannot be initiated from a remote user to a private cluster node. Furthermore, the larger organization the more conservative to allow a new port to pass through the firewall only for grid-computing users.

In order to alleviate these limitations, we have paid attention to such a reasonable point that HTTP is generally permitted to pass through the firewall and that each cluster invokes a web server at its gateway node. Using HTTP tunneling technology, we are designing a TCP proxy in charge of encoding TCP packets into HTTP requests, forwarding them to a remote cluster, decoding HTTP requests into TCP packets, and delivering them to a local cluster node. Our proxy server is a FastCGI process invoked at each cluster gateway. Once initialized by a web server, unlike CGIs and sevlets, it keeps running without being reinvoked [13]. The proxy server uses an IP address translation table to distinguish each remote cluster node by the following three properties of its local cluster gateway: (1) a port, (2) one of private addresses allocated to it, and (3) the number of *rsh*s requested to it. Table 1 shows such a translation table example when two clusters communicate with each other. For instance, assuming that one cluster's gateway node has two private IPs such as *192.168.1.1* and *192.168.1.5*, TCP packets sent to Port *5001* of IP *192.168.1.5* are forwarded to the private node, Port *5000* of IP*192.168.1.4* of the other cluster whose gateway IP is *111.222.333.444*. Whenever a user job needs those two clusters for its computation, (in other words, whenever a user mobile agent reserves them), it should update the translation table of each TCP proxy, so that any nodes in those clusters can transparently communicate with each other using TCP-oriented sockets.

Figure 5 illustrates inter-cluster communication using our TCP proxy server. The sequential bold numbers in the figure correspond to consecutive steps necessary to establish a TCP connection between two clusters. A client user process requests a socket connection to its local gateway's proxy server (step 1), which then creates a new thread to encode the socket request into an HTTP request and to send it to the destination cluster (step 2). The destination web

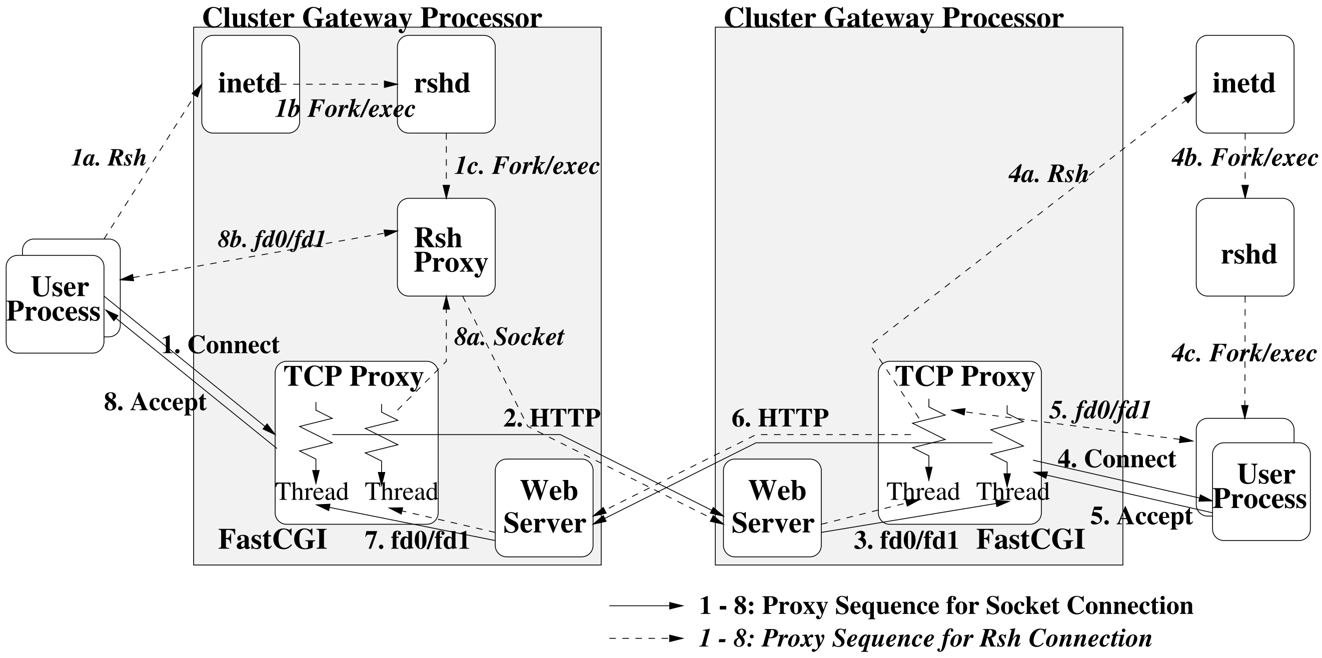

Figure 5. TCP/rsh proxies

server forwards it to its local proxy server (step 3), which allocates a new thread to request a socket connection to the final destination process (step 4). That user process accepts this socket request (step 5), and the acknowledgment returns the opposite way to the client user process (steps 6-8). Needless to say, if there are no firewalls between two clusters, the proxy servers bypass the HTTP encoding/decoding and directly exchange TCP packets with each other for better performance.

Since MPI applications use *rsh* for spawning child processes, we also need to take care of *rsh* between two distinct clusters. Every *rsh* request is first trapped by the *inetd* daemon immediately spawning an *rshd* which then executes a requested program as an independent process. It does not make sense to hack *inetd* directly in order to forward *rsh* requests to a remote cluster. Instead, we instruct *inetd* to spawn our *rsh* proxy rather than the common *rsh* process. This is simply achieved by modifying the */etc/inetd.conf* file. An inter-cluster *rsh* connection is established in quite a similar sequence of the TCP connection. (See its steps ordered in sequential italic numbers in Figure 5.) Only one additional work is to have the TCP proxy server forward every *rsh* acknowledgment to its corresponding *rsh* proxy (step 8a), so that the client user process is fooled into believing the *rsh* proxy as its corresponding remote program.

Obviously, security and performance are the major concerns in HTTP tunneling technology. In terms of security, our coordination environment is supposed to work over trusted clusters. Therefore, our TCP/*rsh* proxy server should be implemented to accept only TCP/*rsh* requests sent to the gateway node's private IP, (i.e., those coming from the local cluster nodes), and HTTP requests arriving from the trusted clusters. In terms of performance, although HTTP encoding/decoding overheads would deteriorate both communication latency and throughput, we expect that the throughput could be improved by aggregating into one bulk HTTP request all TCP packets to be sent to the same remote cluster.

5 Extended Services

Our primary implementation described in Section 4.1 could be still substituted with conventional shell scripts or RPCs rather than realized with mobile agents. However, our ultimate goal is to facilitate the change of intelligent coordination features at run time more elegantly with mobile agents, namely: (1) avoiding faulty, busy, and out-of-service clusters, (2) check-pointing and recovering on-going distributed computation, and (3) dynamic load-balancing.

The key technique is to use the evolvable feature of mobile agents. Such evolution is feasible with loading into agents or replacing their sub-objects with new classes of objects. In addition, remote servers can be also updated by agents carrying and attaching new objects to them. For instance let us revisit the agent code shown in Figure 3. The user mobile agent cannot complete its job if the *server.negotiate* method rejects its request (line 17). However, what if the agent obtains and executes new code inside this method to visit another cluster? When returning from *server.negotiate*, the user mobile agent will be ready to execute its job at the new cluster. Obviously, the main advantage is to relieve users from having to design their mobile agents to tackle all anticipated problems, (an extremely difficult task.)

There are a few evolvable mobile agent systems in use today. For example MESSENGERS mobile agents [1] are capable of dynamically allocating and carrying with them system-provided serializable data regions, termed *dobj*s [6]. The uniqueness of *dobj* regions is that an agent can load new code in, interpret, and detach those regions where it has visited on course of its activity. Voyager implements the same feature in its dynamic aggregation [12], where agents can attach to themselves special objects, called *facet*s, to contain new class of objects, (but detach those *facet*s once attached.) Due to its popularity (in great part for using Java as programming language), we have chosen Voyager in order to implement job coordination adaptable to the dynamic network environment.

Figure 6 shows how a user mobile agent using a Voyager facet can obtain and execute new code to visit another cluster when it has failed in a resource reservation. (Note that this figure shows only new code to be added to the original *UserAg* user mobile agent and *ClusterAg* per-cluster agent.)

UserAg replaces its *coordinate* method with the new version listed in lines 2-9. The new *coordinate* method, immediately after an agent migration, calls the per-cluster agent's *coord* method to attach to itself a new facet implementing the *ICoord* interface (line 3). This facet's *negotiate* method achieves not only resource reservation but also agent migration when the reservation has been rejected. Therefore, when returning from this *negotiate* method, *UserAg* is guaranteed to execute its user job at a new cluster.

On the other hand, *ClusterAg* adds to itself the *coord* method to provide *UserAg* with a *ICoord* facet (lines 12-16). It no longer negotiates with *UserAg* about resource reservation, since that work is performed in the facet's *negotiate* method.

The *ICoord* facet's interface (lines 19-20) must be compiled with both *UserAg* and *ClusterAg*, however *UserAg* does not have to include the actual class implementation, *Coord* (lines 23-30). The agent designer is hence exempted from programming *UserAg* to cope with failure in resource reservation. The *Coord* facet implementation takes care of it, migrates the calling *UserAg* to a next available cluster, and invokes the agent's *coordinate* method again (lines 27-28). A remarkable point is that, as far as *UserAg* receives a new facet from each *ClusterAg* it visits, it will repeat the same sequence of resource reservation and migration until completing the reservation.

We have confirmed this evolvable functionality. As the

```
 (1) public class UserAg .... {
 (2)  public void coordinate(IClusterAg server){
 (3)   ICoord c = server.coord(this);
 (4)    if (c.negotiate(this, reqDOM, server)) {
 (5)     server.exec(job);
 (6)     result=server.getResult();
 (7)    }
 (8)    Agent.of(this).moveTo( home,"report");
 (9)  }
(10)  ....;
(11) }

(12) public class ClusterAg .... {
(13)  public ICoord coord(Object object) {
(14)   IFacets facets=(ICoord) Facets.of(object);
(15)   return (ICoord) faces.of("server.ICoord")
(16)  }
(17)  ....;
(18) }

(19) public Interface ICoord {
(20)  public bool negotiate(Object agent,
(21)           Document reqDOM, IClusterAg srv);
(22) }

(23) public class Coord
(24)           implements ICoord, Serializable {
(25)  public bool negotiate(Object agent,
(26)           Document reqDOM, IClusterAg srv) {
(27)   if (!srv.resource.resource_check(reqDOM))
(28)    Agent.of(agent).moveTo(next,"coordinate");
(29)   return true;
(30) } }
```

Figure 6. Negotiate method made evolvable using a facet

next step, by applying this facet programming, we are planning to implement the following three advanced coordination features:

- *Rerouting:* Prior to shutdown, a cluster server deploys a system mobile agent to its neighbors in order to distribute a new object that allows user mobile agents to avoid this non-service server.

- *Check pointing:* When a user mobile agent intends to visit an error-prone remote server, the current server temporarily saves a copy of this agent in case it needs to resume it upon an abnormal termination. The agent also automatically includes a new object to spawn a system mobile agent to return to the current server to cancel the temporary copy if a successfully job execution occurs.

- *Load balancing:* When unexpectedly overloaded, a server embeds heavy user mobile agents into Voyager's *facet*s and dispatches a system mobile agent to carry them with it and extract the user agents at idle servers.

6 Conclusions

In this paper we have focused on small scale inter-cluster job coordination rather than on a grid-computing environment. The key to such coordination is to offload user coordination work and upgrade the features of remote clusters using mobile agents. We have implemented the first prototype using popular technologies such as Voyager mobile agents, servlets, and XML, and have tested the functionality over clusters located at University of Tsukuba.

Our next goal is to extend our infrastructure to UC Irvine and Ehime University. Needless to say, all clusters among those three institutions are not directly accessible because of some obstacle such as firewalls or private IP addresses. In order to make them temporarily visible to user programs during execution, we must implement our own TCP and *rsh* proxy servers. After that is completed, we will advance our development work to provide extended services using the evolvable feature of mobile agents.

References

[1] L. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, Vol.29(No.8):55–61, Aug. 1996.

[2] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 7.3, Case Study: NetSolve, pages 171–176. Morgan Kaufmann, San Francisco, CA, 1999.

[3] DeleGate Home Page. http://www.delegate.org/delegate/.

[4] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP'98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[5] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[6] M. Fukuda, N. Suzuki, and L. F. Bic. Introducing dynamic data structure to mobile agents. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'99*, pages 1854–1860, Las Vegas, NV, 1999.

[7] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, Vol.32(No.5):29–37, May 1999.

[8] S. Hariri, M. Djunaedi, Y. Kim, R. P. Nellipudi, A. K. Rajagopalan, P. vdlamani, and Y. Zhang. CATALINA: A smart application control and management environment. In *2nd Int'l Workshop on Active Middleware Services*, 2000.

[9] IP Masquerade for Linux. http://ipmasq.cjb.net/.

[10] H. Maruyama, K. Tamura, and N. Uramoto. *XML and Java: Developing Web Applications*. Addison-Wesley Pub Co, 1999.

[11] Ninf Administration Group. Ninf: A network based information library for global world-wide computing infrastructure. Technical note, Electro-Technical Laboratory, Tsukuba, Ibaraki, Japan.

[12] Objectspace. Voyager orb 3.3. Developer guide, Dallas, TX, 1999.

[13] Open Market, Inc. FastCGI: A high-performance web server interface. Technical white paper, http://www.fastcgi.com/, April 1996.

[14] Squid Web Proxy Cache. http://www.squid-cache.org/.

[15] Y. Tanaka, M. Sato, M. Hirano, H. Nakada, and S. Sekiguchi. Resource manager for globus-based wide-area cluster computing. In *Proc. 1st IEEE International Workshop on Cluster Computing*, pages 237–244, Melbourne, Australia, 2-3 December 1999. IEEE-CS.

[16] O. Tomarchio, L. Vita, and A. Puliafito. Active monitoring in grid environments using mobile agent technology. In *2nd Int'l Workshop on Active Middleware Services*, 2000.