A Dynamic Grid Services Deployment Mechanism for On-Demand Resource Provisioning

Eun-Kyu Byun† Jae-Wan Jang† Wook Jung† Jin-Soo Kim‡

Division of Computer Science, Department of EECS,
Korea Advanced Institute of Science and Technology (KAIST)

†{ekbyun, jwjang, wjung}@camars.kaist.ac.kr ‡jinsoo@cs.kaist.ac.kr

Abstract

Recently Grid computing has started to leverage Web services technology by proposing OGSI-standard. OGSI standard defines the Grid service which presents unified interfaces to every participant of Grid. In current Glubus Toolkit3(GT3),which is an implementation of OGSI, Grid service factories should be deployed manually into resources to provide grid services. However, it is necessary to dynamically allocate proper amount of resource, since the demand for resource of service provider changes over time.

In this paper, we propose a architecture to enable on-demand resource provisioning. We develop Universal Factory Service (UFS) that provides a dynamic Grid service deployment mechanism and a resource broker called Door service. Through the experiments, we show that Grid services can adaptively exploit resources according to the request rates.

1. Introduction

Grid computing [1] is a new approach to exploit distributed, heterogeneous, and geographically scattered resources in order to solve complex problems in scientific and commercial areas. The ultimate goal of Grid computing is to create virtual organizations (VOs) through secure, coordinated resource sharing among Grid participants and to provide mechanisms for users to use Grid resources without knowing the detailed characteristics of them.

Recently Grid community has proposed OGSA (Open Grid Service Architecture) [2], in which Grid resources are wrapped as Grid services with standard Web services [5] interfaces to provide unified access methods to users. Standard interfaces described in OGSA are based on OGSI (Open Grid Service Infrastructure) [3]. OGSI provides mechanisms for 1) creating, naming, and discovering Grid service instances, 2) managing Grid service lifetime, and 3) subscribing and notifying specific service data. Hosting environment defined in OGSI and existing well-developed Web services tools make the development of Grid services easier than ever.

Globus Toolkit 3.0 (GT3) [4] is one of the reference implementations of OGSI specification. GT3 consists of GT3 core and several packages including security, data management, resource management, and information services. Using these packages, Grid service providers or users can exploit Grid resources successfully, and Grid resource providers can supply their resources with uniform interfaces. Service providers can release their services by deploying those on the Grid resources. Users of Grid then use remote Grid resources by using Grid service. In the current implementation of GT3, service factory of a service has to be deployed in the resources in order to process service requests. However, the deployment of service factory requires restart of service container by resource administrator. This means that other grid services executed on the resource have to undergo unexpected interruption. That is to say, service providers can not dynamically exploit Grid resources with current GT3.

In this paper, we design and implement a dynamic Grid service deployment mechanism named *Universal Factory Service (UFS)* and a resource broker named *Door service*. With these two service, we suggest a dynamic resource provisioning architecture for service providers. We evaluate UFS and Door service on a prototype Grid testbed. As a result, we show that grid service can be deployed and provides without break of resource availability. We also show that grid service can exploit proper amount of Grid resources adaptively according to the request rates.

0-7803-9074-1/05/$20.00 ©2005 IEEE

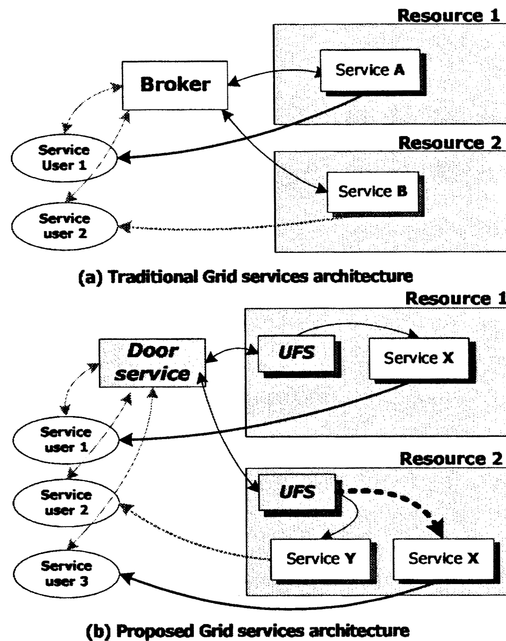

Figure 1. A comparison of traditional and proposed Grid services architecture

2. Motivations

We assume that the following three entities are involved in Grid computing.

- *Service providers* are who develop Grid services and want to deploy the developed Grid services into resources.
- *Resource providers* are who provide resources such as cluster, desktop PCs, workstations, storages and so on.
- *Service users* are who want to use the services developed by service providers.

Since various services have different characteristics, their resource demands vary dynamically. For example, a on-line shopping mall has a huge peak demand in December where majority of its sales occurs. The service provider for the shopping mall should be able to provide resources enough to handle such a peak demand. However, this will lead to significant squandering of resources during other non-peak season. If the service provider can adjust the amount of resources according to the demand, he need not hold resources just to deal with peak periods.

Requirements for on-demand resource provisioning are as follows:

- *Finding resources to meet service providers' demand.* Many resource broker systems [6][7][8][9][10] can choose resources which satisfy the demand of service providers.
- *Deploying services to the selected resources.* In the current GT3, resource providers have to deploy services by hand.
- *Dispatching incoming requests from service users.* Incoming service requests need to be spread out on multiple resources where the services are deployed already.
- *Removing the service when it is idle for a long time.* If the service request rates decreases, deployed services can be removed in order to save disk space and to reduce management costs.

GT3 requires that a certain service factory has to be deployed in the resources in order to serve service users. For example, as shown in 1(a), Service A is only be served on Resource 1 and Service B is only served in Resource 2. In current GT3, deploying any Grid services requires the modification of configuration files and restarting of the GT3 container itself. Thus, it is hard to use othe r free resources for the overloaded services because manual arrangement is unavoidable to deploy the service.

However, in our proposed architecture shown in Figure 1(b), UFS can deploy any kinds of Grid services without modifying the configuration manually and restarting the GT3 container. For example, although *resource 2* does not contain *service X* before, UFS in *resource 2* can deploy *service X* without aforementioned tasks.

In addition, Door service in the proposed architecture dispatches incoming requests to the resources and deploys services into new resources according to the deploying policies. In the previous example, if Door service notices that *service X* is too overloaded to be served only in the *resource 1*, Door service deploys *service X* into *resource 2* which is selected by deploying policies. Moreover, Door service can remove the idle service from unnecessary resources.

With the help of UFS and Door service, service providers can exploit on-demand resource provisioning for unexpected service users' demand. The benefits of using UFS and Door service are summarized as follows:

- *Little effort to deploy services for resource providers.* Resource providers only need to deploy UFS as a Grid service and service providers can deploy their services on-demand if they have appropriate access rights.
- *Improved manageability.* Management cost of Grid resources is substantially high due to the huge scale of

Grid. Since UFS can deploy and remove Grid service dynamically, resource providers need not perform these tasks by hand.

- *Implicit load balancing on multiple resources.* Door service can adjust the amount of resources to guarantee a certain level of quality-of-service to service users without involvement of service providers. Thus, service providers can focus only on their services.

The remaining sections are organized as follows. Section 3 overviews related work. In section 4 we describe the design and implementation of UFS and Door service. Section 5 displays the evaluation results. We present conclusions and future work in section 6.

3. Related work

Various dynamic service deployment mechanisms have been explored in the area of distributed systems.

Océano [11] is a prototype system of a highly available, scalable, and manageable infrastructure for an e-business computing utility. Océano divides server farms into several separate domains, which are completely isolated each other. Each domain provides a hosting environment for one service. When one of the hosted services is overloaded, Océano increases the size of associated domain by deploying the services to free resources dynamically. If another busy service becomes idle, adversely, they decrease the size of that domain by removing the services dynamically. In Océano, preparing another hosting environment for overloaded services takes relatively long time due to the restarting and the reconfiguration of the server in order to meet the requirements of the new services. Thus, adjusting the size of a service domain has to be made carefully.

SODA [12] is another kind of the service platform which supports dynamic service deployment. SODA introduce HUP (Hosting Utility Platform) which is similar to our UFS. HUPs are installed in physical hosts and each host runs one or more virtual service nodes. A service can be installed in one or more these virtual service nodes, which are complete virtual machines on real hosts. With HUP, SODA can support mechanisms for adjusting the amount of virtual service nodes according to the service request rate. If the number of incoming service requests of a service increases, service providers ask SODA agent a new resource, and SODA agent and master allocate more resources to that service, if possible. However, since SODA is based on the virtual machine to provide hosting environment for a service, the cost of changing the amount of resources is considerably high, and thus this decision has to be made cautiously like Océano.

OGSI.NET [13] is an implementation of the OGSI specification on Microsoft's .NET platform. It provides a container framework on which OGSI-compliant Grid computing is supported in the .NET/Windows world. In OGSI.NET architecture, *dispatcher* is similar to our Door service, which routes incoming requests to pertinent Grid Service Wrapper (GSW). While *dispatcher* in OGSI.NET is a local agent routing the incoming service requests to one of GSWs, Door service is a global agent to link incoming service requests to resources. One of other components, metafactory in OGSI.NET, is comparable to our UFS. Metafactory allows service providers to deploy Grid services into the GSW without restarting or reconfiguring OGSI.NET container. However, service codes deployable into GSW are called assembly, which is a different type of the service implementation from GT3.

The GridWeaver project [17] presents an autonomous resource configuration framework for clusters by combining LCFG [19] and SmartFrog [20]. Since these technology are designed for cluster scales, GridWeaver need to be modified to deal with huge resource pools like Grid.

Distributed Ant(DistAnt) [18] provides a mechanism for deploying user's application on remote resource. DistAnt adopts similar deployment mechanism used in UFS. However, DistAnt only focuses on deploying and instantiating custom applications, and does not consider the resource provisioning for Grid-service provider.

Our work primarily differs from previous researches in that we focus on on-demand resource provisioning with the dynamic Grid service deployment mechanism for *OGSI-compliant Grid services.*

4. Design and implementation of UFS and Door service

4.1. Architecture for on-demand resource provisioning

Figure 2 shows the overall architecture of UFS and Door service with relevant entities such as service users, service providers, and resource providers. Since this architecture follows the standard OGSI specification, in order to use a particular service, service users must interact with a service instance created by Grid service factory of that service. The role of each service is as follows:

- *Actual service.* Actual service is a Grid service written in Java that service providers want to provide to service users. Actual service can be deployed to any resources in which UFS is deployed.

- *Universal Factory Service (UFS).* UFS is deployed in every resource. UFS allows service providers to deploy their Grid services dynamically and to create instances of that Grid service.

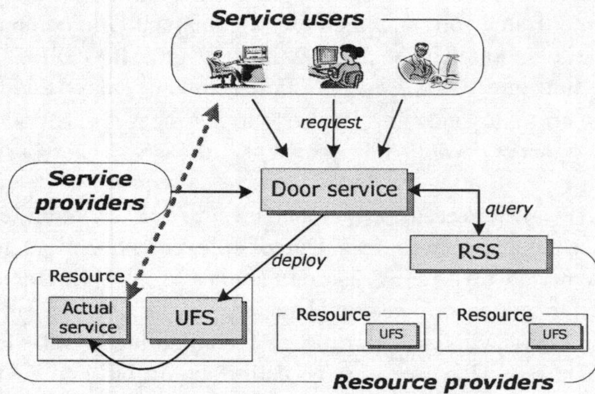

Figure 2. The relationship among service providers, service users and resource providers

- *Resource Selection Service (RSS).* RSS is a kind of Index service which monitors and serves the information of Grid resources. Based on this information, Door service obtains the list of proper resources which satisfies the condition for executing actual service.

- *Door service.* Door service relays service requests to proper resources in which actual service is deployed. Door service also deploys the actual service to the resource with the help of RSS and UFS in order to adjust the amount of resources occupied by the service.

Service users should know the address of GSH(Grid Service Handle) of the service factory in order to create and use the service instances. In our on-demand resource provisioning architecture, the location of the resources where the Grid service is deployed is dynamically changed. Thus, some mechanism to connect the service user to the resources is necessary. Door service takes a part of this mechanism in our architecture.

Note that service providers should have their own Door services to provide their Grid services to service users. We implemented a general Door service so that service providers can use that after simple configuration. Service users then can use the Grid service simply by contacting its own Door service.

Next two subsections explain UFS and Door service in detail. At the end of this section, we show an example scenario how a Grid service is called by a service user.

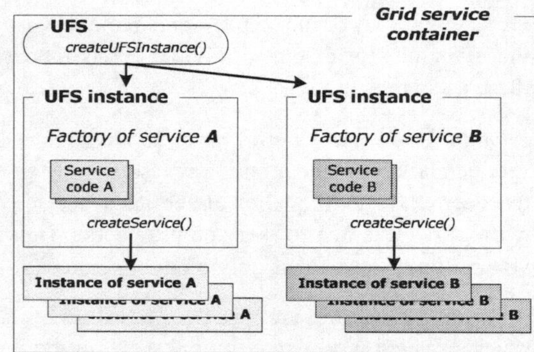

Figure 3. Internal architecture of UFS

4.2. Universal Factory Service (UFS)

Figure 3 depicts how UFS creates service instances. UFS is a persistent Grid service which is permanently accessible as long as the GT3 container is alive. If Door service asks UFS to deploy a new Grid service, UFS creates an UFS instance, which is an empty service at that time. UFS instances can work as a service factory of any Grid service after Door service transfers service code and configures necessary parameters. Since Grid services are implemented as Java classes, no complicated installation procedures are required. Java classes are transferred over http protocol and stored in preassigned local storage. Then just after linking the location of that Java classes to the UFS instance, service users can create service instances using *createService()* method of the UFS instance in the same way they access common Grid service factory.

Since UFS instances support the notification of its resource status, Door service subscribes that resource status. Currently, a UFS instance notifies Door Service of the CPU load average and the number of active service instances.

A UFS instance can be terminated by the *destroy()* method. The *destroy()* method deletes the service code and destroys the UFS instance as other Grid service instances are destroyed.

4.3. Door service

Door service provides service users with the same interface as common Grid service factory does. Service users can get GSH of a service instance from Door service via *createService()* method. To handle the request from service users, Door service asks the proper resource to create a service instance. Door service can deploy the actual service into new resources using UFS, if necessary.

The service providers need to specify the policies including where new service instance should be crested, when the

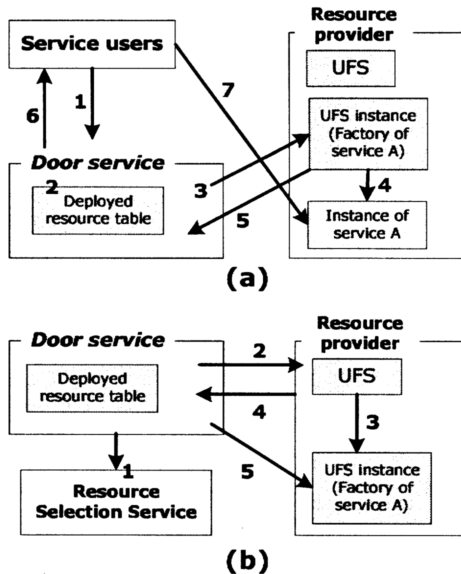

Figure 4. Service request flows

service should be deployed into a new resource, and when the resource should be released. In the current version of Door service, service providers can determine the policy according to the CPU load average and the number of active instances in the resource. Door service maintains a list of GSHs of UFS instances where the actual service is deployed and the status of the resources. The status of resource is updated by notification from UFS instances.

Since UFS instances are not persistent Grid services, dynamically deployed services are no more available if GT3 container is restarted or failed. Nonetheless, since Door service always monitors the status of resource, Door service can detect the resource failure and allocate new resource if the number of remaining resources are insufficient to deal with next requests.

Now we show the example scenario of using Door service and UFS. Assume that s service user want to invoke a Grid service A. Figure 4(a) depicts the operation flows when Door service does not need to deploy the service to new resource. In this case, the following steps are required:

1) A service user asks Door service to create an instance of the service A.

2) Door service selects a resource from the resource pool where the service A is already deployed.

3) Door service sends a request to the UFS instance of the selected resource to create an instance of service A.

4),5) The UFS instance creates a new instance of the service A and returns its GSH to Door service.

6) Door service returns that GSH to the service user.

7) The service user contacts the instance of the service A using the returned GSH.

Figure 4(b) displays the request flow when Door service needs to deploy service A into a new resource. In this case, the following steps should be done between step 2) and 3) of Figure 4(a):

1) Door service gets the list of available resources for the service A from RSS.

2) Door service asks UFS to create a UFS instance.

3),4) UFS creates an UFS instance which is an empty factory and returns GSH of the created UFS instance.

5) Door service transfers the service code of the service A and configures the UFS instance. Then the UFS instance is ready to act as a factory of the service A.

5. Evalution

5.1. Experimental setup

For the experiments, we use eight two-way SMP nodes with Intel Pentium Xeon 2.8 GHz and 1GB memory, and four two-way SMP nodes with Intel Pentium III 850MHz and 1.5GB memory. They are connected by 100Mbps Ethernet. Globus toolkit version 3.2.1 is installed in every node. Door service and RSS are deployed on a Xeon node, while another Xeon node is dedicated to periodical generation of service requests. The remaining ten nodes are used as resources of the resource provider and UFS is installed in each resource.

We convert three real applications, Raja[14], JIU[15], and Jspeex[16], into Grid services. During the experiments, we have varied the rate of service requests to these services.

5.2. Overhead of using Door service

While UFS and Door service provide a convenient way to use resources for service providers, service users may experience an additional delay compared to the case they use the resource directly, since every request should pass through Door service. This delay can be longer when Door service has to deploy the service to a new resource. In our experiments, deploying a service of 4.5Mbytes code through UFS takes 3.98 seconds on average. Notice that transferring the service code over the network consumes 97% out of 3.98 seconds. Since service deployment does not happen frequently, the delay due to the presence of Door service is not critical.

Table 1. The elapsed time for getting a service instance from a resource

Without stub code loading		With stub code loading	
Direct to factory	via Door service	Direct to factory	via Door service
98.47 ms	108.79 ms	3482.65 ms	2573.05 ms

We measured the average round trip time of *createService()* method of Door service. Service users ask Door service to create an instance of Raja Grid service. The result is presented in Table 1. When service users request the Grid service for the first time, service users may wait additional time to get results since they have to load stub codes. This overhead does not appear once stub codes are loaded.

In Table 1, the time without stub code loading shows the delayed time due to Door service. The service users wait about 10 more milliseconds to get results when they request through Door service than when they directly request to factory. This is ignorable compared to the whole execution time of the service.

When we measure the round trip time with stub code loading, *via Door service* takes smaller time than *Direct to factory*. This is caused by the difference in stub code size between two approaches. In the current GT3, service users have to load several classes to access common Grid service factory, while service users should load only Door service related classes in order to access Door service.

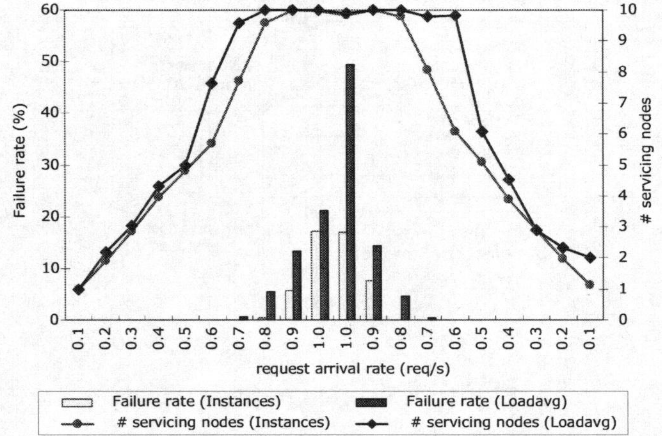

Figure 5. Changes on the number of servicing nodes and request failure rates when service requests vary

5.3. Behavior of UFS and Door service with single Grid service

In order to show our framework provides resources dynamically when the number of requests from service users changes, we vary the arrival rate of requests and observe the number of servicing nodes. Service users periodically make a request to run Raja Grid service which takes a few tens of seconds.

We vary the arrival rate of requests of service users by 0.1 req/s every ten minutes. First, we increase the arrival rate from 0.1 req/s to 1.0 req/s and, after that, decrease from 1.0 req/s to 0.1 req/s. We repeat this experiment for two different policies of deploying/scheduling. The first policy is that Door service does not allow the number of active instances of every deployed resource to exceed the number of processors of the node. On the other hand, in the second policy, Door service does not allow the CPU load average of every deployed resource to exceed the number of processors of the node. In both case, Door service releases the resource when it has no active instance.

Figure 5 presents the number of servicing nodes along with request failure rates. The results of the first policy is denoted as *instances*, and that of the second policy as *loadavg*. Figure 5 shows that Door service utilizes resources proportional to the arrival rates. Services are completely provided to service user until there is no more resource. When the arrival rate is more than 0.8 req/s, some requests are not served because all resources are busy and the policy does not allow to create a new service instance on existing resources.

The result obviously exhibits that *loadavg* occupies slightly more resources with higher failure rate than *instances*. This is caused by the freshness of information Door service uses. While the number of active instances is reported to Door service whenever changed, the CPU load average is reported in every 10 seconds. Besides, the CPU load average denotes the average running processes during last one minute. Therefore, once Door service perceives the resource is busy, Door service may not use that resource even though the resource is available.

5.4. Behavior of UFS and Door service with several Grid services

We perform another experiment to show the behavior of UFS and Door service when several services co-exist on the resources. We deployed three Door services of Raja, JIU,

Table 2. The amount of the service requests normalized to the phase 1

	Phase 1	Phase 2	Phase 3	Phase 4
Raja	1.0	2.0	1.0	0.0
JIU	1.0	0.5	1.5	1.5
Jspeex	1.0	0.5	0.5	1.5

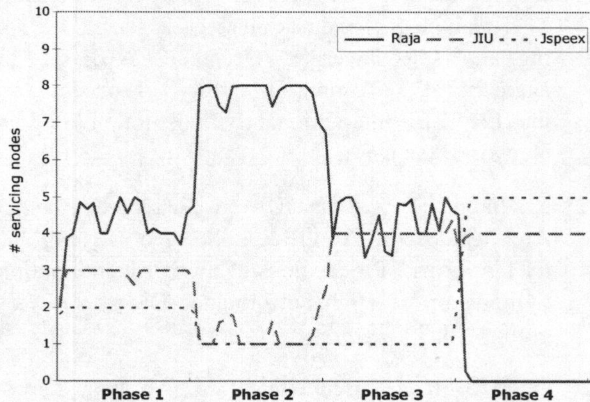

Figure 6. The dynamic change in the number of servicing nodes for three different Grid services

and Jspeex Grid service. Similar to the former experiments, service users periodically make service requests.

We divide the whole experiment time into four phases. Each phase lasts for 20 minutes. In the first phase, service users make total 400 requests for Raja, 600 requests for JIU, and 800 requests for Jspeex. We change the arrival rate of requests whenever the phase is changed. Table 2 shows the relative arrival rates of each phase normalized to those of phase 1. In this experiment, each service adopts the *instances* policy. Figure 6 shows the number of servicing nodes during the experiments. As expected, each service occupies only the amount of resources that is required to handle the incoming requests.

We also show that on-demand resource provisioning can more efficiently utilize the limited amount of resources than the case where services are statically partitioned to resources. For the static deployment system, we deploy Raja into four nodes, JIU into three nodes, and Jspeex into three nodes prior to starting the experiment. As shown in figure 7, the failure rate of the static deployment system is higher than that of the dynamic deployment system. Dynamic deployment system can exploit any resources, whereas static deployment system use only the fixed number of resources.

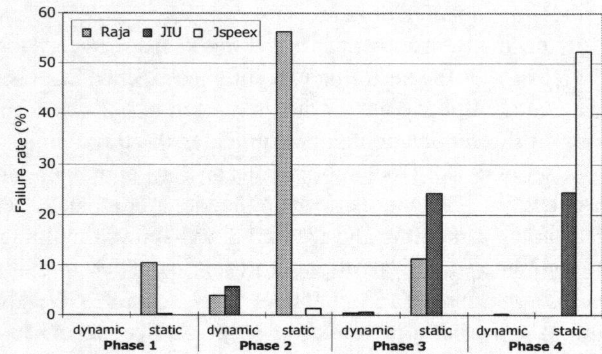

Figure 7. Failure rates experienced by service users

6. Conclusions and future work

In this paper, we identify the requirements for the on-demand resource provisioning for Grid services. Since functionalities of the current GT3 are insufficient to support dynamic Grid service deployment which is essential to support on-demand resource provisioning, we design and implement UFS which is an OGSI-compliant Grid service. Once UFS is deployed into the resources, service providers can deploy any kinds of Grid services without modifying any configuration or restarting the GT3 container. We also design and implement a simple resource broker called Door service. Door service works as a front end of a certain service and dispatches incoming requests of that service to the resources in which actual services are already deployed.

Through the various experiments, we show that our dynamic deployment mechanism works correctly. When the request rate of a service increases, we can observe that the corresponding service occupies more resources. On the other hand, when the request rate of a service decreases, the corresponding service releases its resources and the amount of occupying resources shrinks. We also show that dynamic deployment can use resource more efficiently compared to static partitioning.

Service providers may apply several policies for deploying and dispatching to Door service in order to preserve the reasonable performance of their services. These policies should be determined considering the performance characteristics of a service, the capability of a resource, and the QoS expectation of service provider. The current version of Door service only allows service provider to limit the number of concurrent instances according to the number of processors. However, since the characteristics of many Grid services vary, our next version of Door service will support additional mechanisms for service providers to de-

scribe more detailed policies.

If too many users want to use the same service, Door service may be the performance bottleneck. Since Door service is quite lightweight as shown in section 5.2, the scalability of the service container dominates the throughput of Door service. Our Door service runs on GT3 container. Unfortunately, GT3 container can not handle more than 20 services at the same time and we find that it shows the limited throughput even for very light Grid service. Therefore, it is necessary to make GT3 container more scalable. We also plan to construct a hierarchy of Door services to enhance the scalability of Door service.

References

[1] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," International Journal of Supercomputer Applications, 2001.

[2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[3] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling, "Open Grid Services Infrastructure (OGSI) Version 1.0," Global Grid Forum Draft Recommendation, 6/27/2003.

[4] The Globus Alliance, http://www.globus.org/.

[5] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," IEEE Internet Computing, 2002.

[6] C. Liu, L. Yang, I. Foster, and D. Angulo, "Design and Evaluation of a Resource Selection Framework for Grid Applications," Proceedings of the IEEE International Symposium on High-Performance Distributed Computing, 2002.

[7] W. Lee, S. McGough, S. Newhouse, and J. Darlington, "Load-balancing EU-DataGrid Resource Brokers," UK e-Science All Hands Meeting, 2003.

[8] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003.

[9] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," Proceedings of the IEEE International Conference on High Performance Computing and Grid in Asia Pacific Region, 2000.

[10] Y.-S. Kim, J.-L. Yu, J.-G. Hahm, J.-S. Kim, and J.-W. Lee, "Design and Implementation of an OGSI-Compliant Grid Broker Service," Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2004.

[11] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger, "Océano–SLA Based Management of a Computing Utility," Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, 2001.

[12] X. Jiang, D. Xu, "SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms," Proceedings of the IEEE International Symposium on High Performance Distributed Computing, 2003.

[13] G. Wasson, N. Beekwilder, M. Morgan, and M. Humphrey, "OGSI.NET: OGSI-compliance on the .NET Framework," Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2004.

[14] The Raja Project, http://raja.sourceforge.net/.

[15] JIU - The Java Imaging Utilities - An image processing library, http://jiu.sourceforge.net/.

[16] JSpeex - Java Implementation of Speex, http://jspeex.sourceforge.net/.

[17] GridWeaver Project, http://www.gridweaver.org/.

[18] W. Goscinski and D. Abramson, "Distributed Ant: A System to Support Application Deployment in the Grid," IEEE/ACM International Workshop on Grid Computing, 2004.

[19] P. Anderson and A. Scobie, "LCFG - the Next Genaration," UKUUG Winter Conference, 2002.

[20] P. Goldsack, "Smartfrog: Configuration, ignition and management of distributed application," Technical report, HP Resource Labs. http://www-uk.hpl.hp.com/smargfrog/.