Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 33 (2007) 328–338
PARALLEL COMPUTING
www.elsevier.com/locate/parco

DynaGrid: A dynamic service deployment and resource migration framework for WSRF-compliant applications

Eun-Kyu Byun *, Jin-Soo Kim

Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea

Available online 20 February 2007

Abstract

Large-scale Grid is a computing environment composed of Internet-wide distributed resources shared by a number of applications. Although WSRF and Java-based hosting environment can successfully deal with the heterogeneity of resources and the diversity of applications, the current Grid systems have several limitations to support the dynamic nature of large-scale Grid.

This paper proposes DynaGrid, a new framework for building large-scale Grid for WSRF-compliant applications. Compared to the existing Grid systems, DynaGrid provides three new mechanisms: dynamic service deployment, resource migration, and transparent request dispatching. Two core components, ServiceDoor and dynamic service launcher (DSL), have been implemented as WSRF-compliant Web services to realize DynaGrid, which are applicable to any Java-based WSRF hosting environment. We construct a real testbed with DynaGrid on the Globus Toolkit 4 and evaluate the effectiveness of our framework using two practical applications. The evaluation results show that dynamic service deployment and resource migration in DynaGrid bring many advantages to large-scale Grid in terms of performance and reliability with minimal overhead.

1. Introduction

Grid computing is the technology for building Internet-wide computing environment integrating distributed and heterogeneous resources [1]. The ultimate goal of Grid computing is to create a virtual organization (VO) for secure and coordinated resource sharing among Grid participants and to provide standard mechanisms for users to exploit Grid resources in the VO. Open Grid services architecture (OGSA) [2] is a standard Grid architecture which assures interoperability on heterogeneous Grid resources by exploiting Web services protocols. Recently, Web services resource framework (WSRF) [3], a new Web services standard, has been introduced to realize OGSA by the joint efforts of Grid and Web services communities.

* Corresponding author. Tel.: +82 42 869 3569.
E-mail address: ekbyun@camars.kaist.ac.kr (E.-K. Byun).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.02.005

WSRF is a group of specifications which define a generic and open framework for modeling and accessing stateful resources using Web services. A stateful resource is a set of data values that persist across, and evolve as a result of, Web service interactions. Note that the term *resource* used in WSRF should not be confused with the resource in Grid computing, a general term to denote a computational or storage resource. In the rest of this paper, we use the term "*ServiceResource*" to explicitly indicate the stateful *resource* defined in WSRF specifications. A WS-Resource is the composition of a ServiceResource and a Web service through which the ServiceResource can be accessed. Each WS-Resource is identified by an endpoint reference containing the URI of the service and the key of the stateful ServiceResource.

Globus toolkit version 4 (GT4) [4] is a representative WSRF-compliant Grid middleware. GT4 provides a GT4 container which is a Java-based hosting environment for WSRF-compliant Web services and ServiceResources. The GT4 container is executed on each host, and distributed GT4 containers eventually form a VO. A typical step to run an application on a specific host is to implement the application as a WSRF-compliant service and deploy it into the container of the host.

The focus of this paper is to construct an efficient and reliable large-scale Grid, where a number of applications share Internet-wide distributed resources. Especially, we pay attention to the following characteristics of large-scale Grid. First, large-scale Grid is composed of a large number of heterogenous resources on which diverse applications are executed. Second, the amount of resources demanded by an application may change over time. Third, each resource may join or leave the Grid freely, hence the availability of the resources may also vary dynamically.

Although GT4 has been successful in dealing with the heterogeneity of resources and the diversity of applications by the use of platform-independent Web services protocols and Java-based hosting environment, GT4 has several limitations to support the latter two characteristics. The dynamic nature of large-scale Grid requires that new resources can be allocated adaptively to the application which demands more computing power. It should be also possible to migrate the ServiceResource, the state of a running service, to another stable resource when the current resource leaves the system or experiences unstability, overload, or failure. In the current implementation of GT4, however, each container exports only the services that are manually deployed by a local administrator. To deploy a new service, the existing container should be stopped which leads a situation that all the services in the container become unavailable and the stateful ServiceResources are lost. GT4 also lacks a facility that can migrate the ServiceResource to another resource to cope with the varying conditions of a resource.

In this paper, we propose DynaGrid, a new framework for constructing an efficient and reliable large-scale Grid for WSRF-compliant applications. DynaGrid aims at extending GT4 to provide the following three functionalities required for large-scale Grid.

- *Dynamic service deployment*:
  In our framework, services can be dynamically deployed on any resource without the restart of the running container. This allows to maintain a certain level of quality-of-service even if the resource demand of an application fluctuates.
- *Dynamic ServiceResource migration*:
  DynaGrid provides the ability to migrate ServiceResources to another resource dynamically, which is necessary to rescue an overloaded container or to terminate a container safely.
- *Transparent request dispatching*:
  Due to dynamic service deployment and ServiceResource migration, the actual location of a service instance can not be determined statically. Thus, DynaGrid provides a service-specific access point, called ServiceDoor, through which users can access the service instances and ServiceResources in a transparent way.

Our experimental evaluations show that these functionalities can be supported effectively with minimal overhead on the current GT4 implementation. Since DynaGrid is implemented as Web services, it can be used not only with GT4 container but also with any Java-based hosting environment which complies with WSRF.

The rest of the paper is organized as follows. In Section 2, we briefly overview the related work. Section 3 presents the overall architecture and the main functionalities of DynaGrid. Section 4 provides the evaluation results of our framework with two practical applications. Finally, we conclude in Section 5.

2. Related work

PlanetLab [5] is a large-scale distributed platform which integrates Internet-wide resources to obtain massive computing power. Although PlanetLab has been successfully used as a testbed for computer networking and distributed systems research, PlanetLab can be run only on Linux platforms with special configurations and there is no systematic mechanism to deploy a new service or to migrate the existing services.

Several previous studies have dealt with dynamic service deployment mechanisms. Weissman et al. [6] suggested a new Grid architecture which supports dynamic service deployment and dynamic service leasing according to the changes of service demands. Smith et al. [7] proposed a Service-Oriented Ad Hoc Grid which provides peer-to-peer based node discovery, automatic node property assessment, and hot deployment of services into a running system. Our earlier work was also devoted to the development of a dynamic service deployment mechanism for globus toolkit version 3 (GT3) in which universal factory service (UFS) enables to start a new Web service instance without any intervention of the administrator [8]. These solutions, however, have restrictions that they provide dynamic service deployment mechanisms in OGSI-based Grid environments. Open Grid services infrastructure (OGSI) [9] is an extension of Web services specification for implementing OGSA. Since OGSI supports stateful services by introducing a new type of Web service called *Grid service*, OGSI does not comply with Web services standards and thus now it is deprecated in favor of WSRF.

Recently, HAND [10] is proposed as a dynamic service deployment mechanism for the GT4 container and it is known to be included in the later version of GT4. However, since HAND requires a special function which modifies internal data of the GT4 container with the special privilege, its implementation can not be generally applicable to other hosting environments.

Our approach is the first attempt to build large-scale WSRF-compliant Grid systems. DynaGrid effectively handles the dynamic nature of large-scale Grid, the heterogeneity of resources, and the diversity of applications by introducing three new mechanisms such as dynamic service deployment, ServiceResource migration, and transparent request dispatching. Our dynamic service deployment mechanism differs from HAND in that our approach can be applicable to any Java-based hosting environment instead of being a GT4-specific solution. In addition, ServiceResource migration is unique to DynaGrid and, to the best of our knowledge, any similar mechanism has not been investigated for WSRF in the previous work.

3. DynaGrid

3.1. Overall architecture

Fig. 1 shows the overall architecture of large-scale Grid constructed with DynaGrid framework. Two core components of DynaGrid are ServiceDoor and dynamic service launcher (DSL). ServiceDoor is a service-

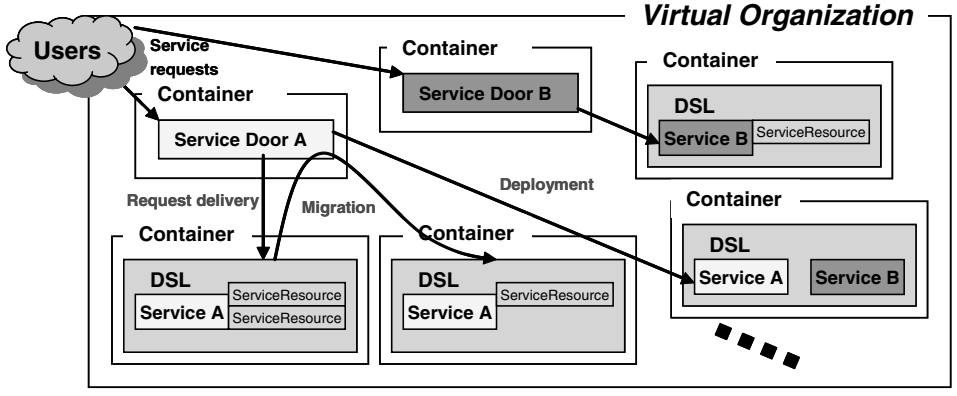

Fig. 1. The overall architecture of DynaGrid.

specific front-end. Every service should have its own ServiceDoor somewhere and its location is published by the register service. Each ServiceDoor keeps track of a list of containers on which the corresponding service is deployed and the user's request is delivered to an appropriate DSL in one of the available containers. ServiceDoors also make decisions on dynamic service deployment and ServiceResource migration and ask DSLs to perform such tasks if necessary. DSL is a passive service controlled by ServiceDoor which actually deploys services, creates ServiceResources, invokes the requested service, and transfers ServiceResource objects for migration.

Since the interface of ServiceDoor is identical to the standard Web service interface, users can access ServiceDoors just like normal Web services. We provide a utility called DoorCreator which automatically creates the ServiceDoor code that is customized to each service from the corresponding service codes such as Web services description language (WSDL) and Web services deployment descriptors (WSDD).

Fig. 2 depicts the internal structures of ServiceDoor and DSL. ServiceDoor is composed of Service PortTypes, Resource manager, service deployment module, Scheduling module, and migration module. Service PortTypes receive and dispatch the user's requests to local DSLs. Resource manager maintains ServiceResource information and a list of containers where the service is deployed. Scheduling module decides the location where a new ServiceResource is created. Service deployment module and migration module automatically carry out dynamic service deployment and ServiceResource migration, respectively, in the background.

DSL consists of DSLResourceHome and a set of methods required for dynamic service deployment, service invocation, and ServiceResource migration. DSLResourceHome is responsible for creating, destroying, and finding Meta ServiceResources and normal ServiceResources. Meta ServiceResource is used to distinguish various services in DSL, while normal ServiceResource is used to preserve the state of each service request as defined in WSRF specifications.

In the following subsections, we explain in more detail how these modules interact each other to achieve transparent request dispatching, dynamic service deployment, and ServiceResource migration.

3.2. Transparent request dispatching

The first step to invoke a service in WSRF is to create a ServiceResource and to obtain the key of the newly created ServiceResource. The key, along with the URI of the service, forms an endpoint reference (EPR) which is used to make a service execution request. Since the exact locations of the service and the ServiceResource are determined dynamically under DynaGrid due to dynamic service deployment and ServiceResource migration, we use ServiceDoors to transparently dispatch incoming requests to the relevant DSLs.

To create a ServiceResource in DynaGrid, a user contacts the PortType of the creation method in ServiceDoor. The creation method invokes Scheduling module to find an appropriate container among the container list kept in Resource manager. If there is no available container in the list, Service deployment module is invoked and the service is deployed to one of idle containers (details will be discussed in Section 3.3).

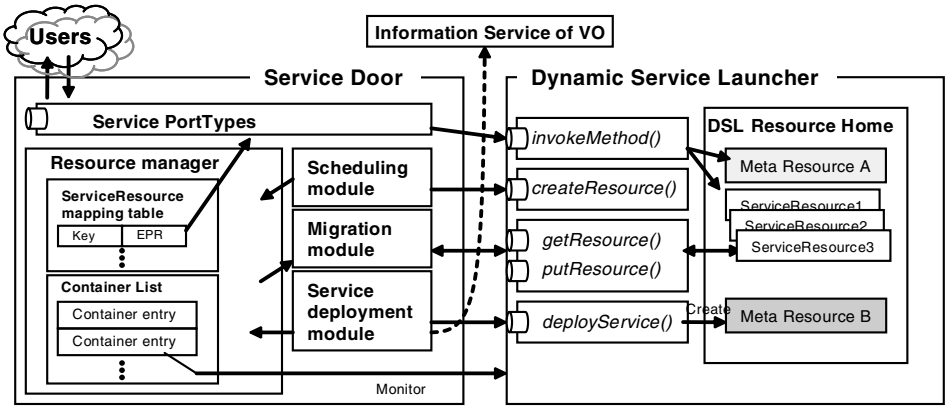

Fig. 2. The internal structures of ServiceDoor (left) and DSL (right).

Scheduling module finally composes an EPR indicating the Meta ServiceResource of the corresponding service in the selected container and uses this EPR to invoke *createResource( )* method of the target DSL. DSL-ResourceHome in the invoked DSL then creates a new ServiceResource according to the information in the Meta ServiceResources and returns a new local EPR containing the key of the newly created ServiceResource. Note that this local EPR can not be delivered to users directly because DynaGrid may dynamically relocate the ServiceResource if necessary. Instead, Scheduling module provides users with a new key associated with the local EPR and maintains a mapping table between them in Resource manager.

The actual service execution request is also sent to ServiceDoor with a key previously obtained from ServiceDoor. Since the information associated with the key is already stored in the mapping table in Resource manager, ServiceDoor can easily retrieve the EPR of the ServiceResource which indicates the current location of the ServiceResource. Using the EPR, ServiceDoor calls *invokeMethod( )* of the target DSL, and as a result of it, the service is executed.

3.3. Dynamic service deployment

Dynamic service deployment is one of the most distinctive features in DynaGrid. In order to compare it with the existing Grid system, we first describe the traditional approach to handling users' requests to execute a service.

Fig. 3a shows how a service is executed in the current GT4 container. First, users ask a GT4 container which hosts the service to create a ServiceResource and obtain the key of the ServiceResource. Second, users issue a service execution request with an EPR composed of the URI of the service and the ServiceResource key. Then the container parses the EPR and finds the corresponding service in the deployed service list. ResourceHome of the service retrieves the corresponding ServiceResource according to the key. Finally, a thread called *context* is started with the ServiceResource.

Deploying a new service into the GT4 container can be done by adding an entry to the container's deployed service list. In the current implementation of GT4, the change in the list is reflected only after the container is restarted. In order to overcome this limitation, HAND [10] proposed a dynamic service deployment mechanism by modifying the GT4 container. It adds an internal function into the container which allows to reload the deployed service list dynamically. However, this approach is not generally applicable to other hosting environments since this is only a GT4-specific solution and requires administrative privilege to access the internal data structure of the container.

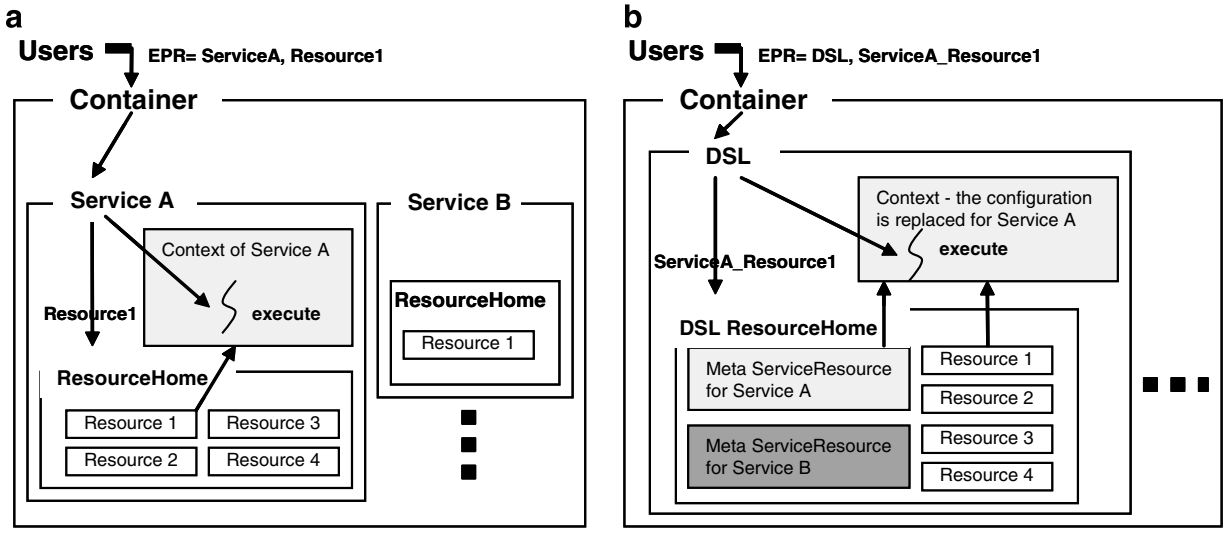

Fig. 3. The service execution mechanisms in (a) GT4 and (b) DynaGrid.

On the contrary, our approach is to devise a dynamic service deployment mechanism which requires no modification in the GT4 container, hence being applicable to any Java-based hosting environment. The mechanism is implemented as a part of DSL which is a standard Web service. We define a new type of ServiceResource named Meta ServiceResource to be used in DSL. When a new service is deployed, a new Meta ServiceResource is created. It keeps the information of the deployed service including the service ID, the interface class, service options, and ClassLoader. Meta ServiceResource is also used for *createResource( )* and *putResource( )* methods, which are service-specific but not related to any ServiceResource. Another role of Meta ServiceResource is to notify the current status of the container to ServiceDoor. This information is later used by ServiceDoor for dynamic deployment or migration decision.

Fig. 3b shows the service execution mechanism in DynaGrid. When the container accepts a request from ServiceDoor, it starts a context to execute the *invokeMethod( )* of DSL with the key composed of both service ID and a local ServiceResource key. DSLResourceHome parses the key to retrieve the Meta ServiceResource related to the service ID and the ServiceResource corresponding to the local ServiceResource key. DSL changes the context's ClassLoader and service options according to the information stored in the Meta ServiceResource in order to configure the context compatible for executing the service. Finally, DSL executes the service with the ServiceResource and returns the result to the user.

In DynaGrid, dynamic service deployment is initiated by invoking Service deployment module in ServiceDoor. There are two cases in which the module is invoked. First, Scheduling module uses Service deployment module to create a new ServiceResource when all the containers currently hosting the service are busy or inaccessible. Second, Migration module uses Service deployment module in case there is no available container to migrate. Dynamic service deployment in DynaGrid works as follows. Service deployment module searches for an idle and stable container using the information service of VO. After finding a suitable container, it transfers the package file containing the service code and ServiceResource class, and deploys the service to the selected container through *deployService( )* of DSL. At the end of the deployment process, DSL returns the EPR of the created Meta ServiceResource. The EPR is inserted in the container list of Resource manager in ServiceDoor.

DynaGrid provides two package file transfer mechanisms. The first one is to transfer a file in a SOAP message as a byte array. The advantage of this approach is that all the steps of service deployment can be done only with standard Web services protocols. This approach, however, increases the transfer time because the overhead of SOAP is considerably high and SOAP encodes a byte array to a base-64 array, which increases the file size by about 30%. In order to overcome such performance degradation, we run a simple Web server in ServiceDoor through which DSL can receive files over HTTP. Even though this approach achieves better performance than the previous one, it has a drawback that an additional network port is required. This can be a problem in an environment where only the SOAP port is allowed to access.

3.4. ServiceResource migration

All of the existing WSRF-compliant systems are designed for each ServiceResource to be handled only on its birthplace since ServiceResource migration is not a standard feature of WSRF specifications. However, since the availability of a container can affect the availability of ServiceResource, we developed a mechanism to dynamically migrate ServiceResource. Again, our approach to ServiceResource migration is independent of the GT4 implementation, and can be used for any Java-based WSRF hosting environment.

Once a service is deployed in a container, Resource manager in ServiceDoor monitors the status of the container. If the container's status meets a predefined condition, the migration is initiated and a half of ServiceResources in the container is migrated to another container. In the current implementation of DynaGrid, service developers can set various migration conditions using the CPU utilization, the used heap size, and the number of ServiceResources described in the WSDD of the service. In addition, DynaGrid unconditionally migrates all the ServiceResources when a container sends a "container stop" notification in order to leave the VO. In this case, Migration module distributes ServiceResources evenly to all the containers in the container list of Resource manager as long as any of the migration condition is not violated. If the ServiceResources can not be accommodated in the existing containers, Service deployment module is invoked and asked to secure new containers for the service.

One of the basic requirements of migrating ServiceResources is that ServiceResources must be serializable Java objects. DSL provides *getResource( )* and *putResource( )* methods for transferring serialized ServiceResources. To make service developers easily implement their ServiceResources, DynaGrid also provides APIs to implement ServiceResources as serializable classes.

The initial step to migrate a ServiceResource is to acquire a migration lock of the ServiceResource to make sure no users are allowed to access it during migration. After successful acquisition of the lock, Migration module obtains the serialized ServiceResource object using *getResource( )* of the source DSL and puts the object to the destination DSL using *putResource( )*. Finally, the ServiceResource lists in DSLResourceHome of both the source and the destination DSL and the mapping table in Resource manager are updated.

DynaGrid also provides mechanisms to subscribe to various topics on the ServiceResource. A topic is an event that notifies the change of data in ServiceResource as defined in WS-Notification of WSRF specifications. DynaGrid guarantees that all the subscriptions are maintained transparently to users even if the ServiceResource is migrated. When a ServiceResource is migrated, all the subscriptions related to the ServiceResource are removed from the source container and re-registered in the new container. Note that ServiceResources have a buffer for topics to prevent loss of notifications during migration.

4. Evaluation

In this section, we show that the overhead incurred by our framework is fairly small and that dynamic service deployment and ServiceResource migration bring many advantages to large-scale Grid particularly in terms of performance and reliability.

4.1. Experimental environment

Our testbed consists of eight servers connected by 100 Mbps Ethernet. Each server is powered by dual 2.4 GHz Pentium 4 processors and 1 GB memory running Linux 2.4.20 and JDK 1.4.2. We use the container of Globus Toolkit 4.0.1 as WSRF-compliant hosting environment.

For our evaluation, we implemented two benchmark applications, *streaming buffer service* and *ray tracing service*, as WSRF-compliant Web services. Their functions and characteristics are as follows.

- *Streaming buffer service* exploits the server's physical memory to buffer streaming data similar to ring buffered network bus (RBNB) [11]. RBNB has been used in many Grid systems including NEESGrid [13] for data aggregation, streaming, and synchronization. The ServiceResource in streaming buffer service is in charge of buffering data generated from a source channel. The service provides methods for storing and fetching buffered data. The size of the ServiceResource in this service can be large up to tens of megabytes.
- *Ray tracing service* is a Web service implementation of RAJA [12], an open source ray tracer. Its ServiceResource maintains a result buffer and a request queue on which scene data and the required quality options are placed. The service continuously executes requests on the queue and stores the result in the ServiceResource. The users are notified when a new result is available to download. Since the service is CPU-intensive, a large number of concurrent executions may increase the response time.

The ServiceDoors corresponding to two benchmarks are automatically generated by DoorCreator. We use two servers for executing ServiceDoors and client programs, and the remaining servers for deploying and executing services through DSL.

4.2. The effectiveness of DynaGrid framework

In the traditional Grid system, computational resources are allocated statically and the services are manually deployed in advance to cope with the peak demand. This results in an inefficient resource utilization and high maintenance cost. Dynamic service deployment can resolve this problem by allocating only the required number of containers on demand according to the amount of incoming requests. We carried out two experiments to show that DynaGrid can utilize the containers in a more effective way.

First, we measure the amount of occupied memory by the streaming buffer service. Fig. 4 shows the change of the aggregated buffer size of 24 streaming channels on which data are generated at 100 KB/s. On each container, the total memory capacity allocated to a buffer is limited to 100 MB. In the case of dynamic service deployment, the service is initially deployed on a single container and all channels are created on it. When the heap of the container is filled with the buffers, the service is deployed on a new container and a half of buffers are migrated. In static deployment cases, the channels are created evenly on the allocated containers. We can observe that, for the dynamic deployment case, the aggregated buffer size is constantly expanded until they occupy most of the heap memory of every container. Meanwhile, the total buffer size is limited in the static allocation cases, where only 1/3 or 2/3 of containers are allocated. The slower growth rate of the aggregated buffer size in the dynamic deployment case is due to the migration cost of ServiceResources.

Second, we compare the average execution time in the ray tracing service as shown in Fig. 5. In this experiment, clients continuously generate the requests according to the poison distribution and the individual request takes about 10 min on a dedicated container. In the dynamic case, the service is deployed to a new container when the CPU utilization exceeds 90%. Fig. 5 illustrates that the variance in the average execution time is kept small when dynamic service deployment is performed. Apparently, the static allocation scheme which exploits only a half of containers does not appear resilient enough to support the high resource demand. Compared to the case which uses all the containers statically, the dynamic scheme slightly increases the average response time due to the overhead in ServiceDoor and DSL, but as can be seen in Fig. 5, the overhead is negligible. The overhead in DynaGrid is analyzed in detail in the following subsections.

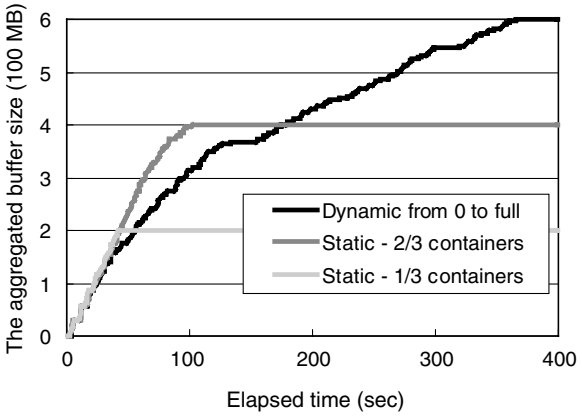

Fig. 4. The aggregated buffer size in the streaming buffer service.

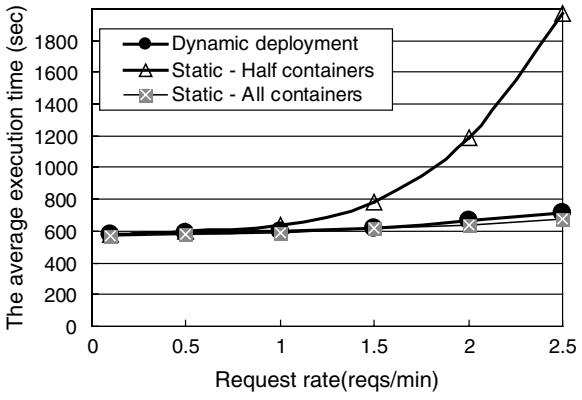

Fig. 5. The average execution times in the ray tracing service.

Table 1
The average latency comparison between direct and indirect (via ServiceDoor) invocation

Service name	Method name	Direct (ms)	via ServiceDoor (ms)	SOAP size (bytes)
Streaming buffer	createChannel()	47	1177	1267
	putData()	67	132	2623
	putData()	171	301	125,345
	putData()	437	927	525,532
	putData()	2410	4914	6,432,914
Ray tracing	createQueue()	41	1259	1252
	addScene()	58	114	14,231

4.3. The overhead of ServiceDoor

The use of ServiceDoor may increase the latency since it requires one more TCP connection and SOAP message parsing. In Table 1, we compare the latencies to invoke several representative methods in our benchmarks either directly or indirectly via ServiceDoor. We can see that using the ServiceDoor approximately doubles the latency compared to the direct invocation for *putData( )* and *addScene( )*. Although the amount of overhead is related to the SOAP message size, the overhead is kept low less than several hundreds of milliseconds unless the SOAP message size is larger than a few megabytes, which is not common in many applications. Note that ServiceDoor adds more than one second to the latencies of *createChannel( )* and *createQueue( )* which are methods for creating new ServiceResources. This is because it takes about one second to find an idle container in Scheduling module of ServiceDoor. Such overhead is not critical since the creation of ServiceResource is performed only once for the first request.

4.4. The overhead of dynamic service deployment

Since ServiceDoor tries to deploy the service to a new container on demand, fast deployment can minimize the overloaded period of a container. Table 2 compares the deployment times of two services. We can see that the time for transferring a service code (represented as "transfer") is a dominant factor, which is directly related to the service code size. The other costs, such as the time for finding an available container and the time for configuring the new container, are small enough and they are nearly constant regardless of the service.

As described in Section 3.3, we provide two file transfer mechanisms: SOAP and HTTP. Fig. 6 presents the performance of two mechanisms in our testbed. As expected, the use of HTTP shows significantly better

Table 2
The comparison of the service deployment time over SOAP (in milliseconds)

Service name	Size (bytes)	Total	Transfer	Find container	Config.	Others
Streaming buffer	61,440	1062	348	529	41	144
Ray tracing	16,777,216	5862	5151	532	41	138

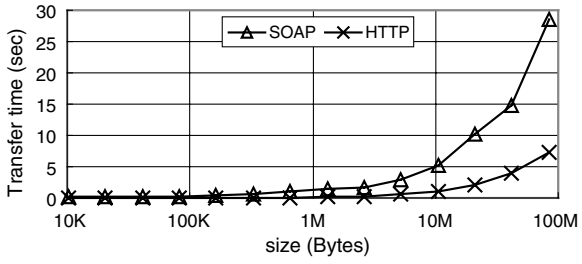

Fig. 6. The performance comparison of SOAP and HTTP for transferring a service code.

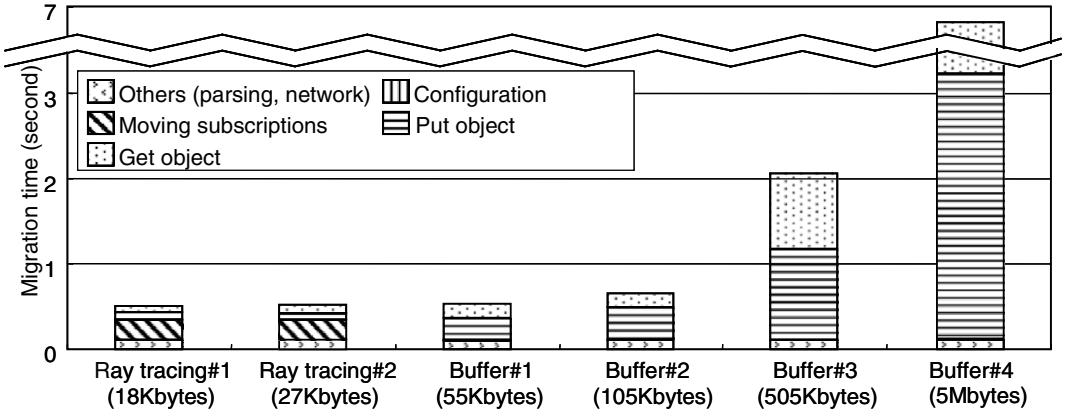

Fig. 7. The breakdown of ServiceResource migration time.

performance compared to SOAP. Therefore, it is desirable to adopt HTTP in order to reduce the service deployment time when the size of the application is large and opening a new port is allowed.

4.5. The overhead of ServiceResource migration

During ServiceResource migration, DynaGrid transfers the serialized object, re-establishes subscriptions, and re-configures DSLResourceHome and Resource manager. In addition, the user's requests are delayed since accessing ServiceResource is not permitted during migration. Fig. 7 shows the breakdown of the migration time with various sizes of ServiceResources. Obviously, the migration time is dominated by the size of ServiceResource object and most of the time is spent for *getResource()* and *putResource()*. Thus, the migration of large ServiceResources should be carefully decided to minimize the increase in the latency. In our streaming buffer service, for example, we select the least frequently accessed channel for the migration.

5. Conclusion

In this paper, we present several limitations of the current WSRF-based Grid system for efficient and reliable resource sharing on large-scale Grid. The heterogeneity of resources and the diversity of applications along with the dynamic nature of the large-scale Grid demands a new Grid computing framework. This paper proposes DynaGrid for building large-scale Grid for WSRF-compliant applications. We adopt WSRF and Java-based hosting environment to overcome the heterogeneity of resources and the diversity of applications. We also developed dynamic service deployment and ServiceResource migration mechanisms for WSRF-compliant applications to handle the dynamic nature of large-scale Grid. Under DynaGrid, applications can exploit various resources dynamically and their running contexts can be migrated to another resource if needed. Two core components, ServiceDoor and DSL, have been implemented to realize DynaGrid, which are compatible with any Java-based hosting environment.

We built a real testbed on GT4 and evaluated DynaGrid with two benchmark applications. Our evaluation shows that dynamic service deployment and ServiceResource migration bring many advantages to large-scale Grid particularly in terms of performance and reliability with minimal overhead.

Acknowledgement

This work was supported by Korea Research Foundation Grant funded by Korea Government (MOEHRD, Basic Research Promotion Fund) (KRF-2005-003-D00291).

References

[1] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: enabling scalable virtual organizations, International Journal of Supercomputer Applications 15 (3) (2001).
[2] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The physiology of the grid: an open grid services architecture for distributed systems integration, Open Grid Service Infrastructure WG, GGF, June 2002.
[3] OASIS Web Services Resource Framework (WSRF) TC, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.
[4] I. Foster, Globus toolkit version 4: software for service-oriented systems, Lecture Notes in Computer Science, vol. 3779, Springer-Verlag, 2005.
[5] L. Peterson, T. Anderson, D. Culler, T. Roscoe, A blueprint for introducing disruptive technology into the internet, in: Proceedings of ACM HotNets-I Workshop, 2002.
[6] J. Weissman, S. Kim, D. England, A framework for dynamic service adaptation in the grid: next generation software program progress report, in: Proceeding of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2005.
[7] M. Smith, T. Friese, B. Freisleben, Towards a service-oriented Ad Hoc grid, in: Proceedings of the 3rd International Symposium on Parallel and Distributed Computing/the 3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar), 2004, pp. 201–208.
[8] E.-K. Byun, J.-W. Jang, W. Jung, J.-S. Kim, A dynamic grid services deployment mechanism for on-demand resource provisioning, in: Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), 2005.
[9] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling, Open grid services infrastructure (OGSI) version 1.0, Global Grid Forum Draft Recommendation, June 27, 2003.
[10] L. Qi, H. Jin, I. Foster, J. Gawor, HAND: highly available dynamic deployment infrastructure for globus toolkit 4, http://www.globus.org/alliance/publications/papers/HAND-Submitted.pdf.
[11] Creare Inc., Ring buffered network bus, http://rbnb.creare.com/RBNB.
[12] The Raja Project, http://raja.sourceforge.net/.
[13] The NEESGrid Project, http://it.nees.org/.