DynaGrid: An Adaptive, Scalable, and Reliable Resource Provisioning Framework for WSRF-Compliant Applications

Eun-Kyu Byun · Jin-Soo Kim

Received: 8 October 2007 / Accepted: 3 July 2008 / Published online: 9 August 2008

Abstract The Web Services Resource Framework (WSRF) is a set of specifications which define a generic and open framework for modeling and accessing stateful resources using Web services. This paper proposes DynaGrid, a new framework for WSRF-compliant applications. Many new components, such as ServiceDoor, Dynamic Service Launcher, ClientProxy, and PartitionManager, have been introduced to offer adaptive, scalable, and reliable resource provisioning. All of these components are implemented as standard WSRF-compliant Web services, hence DynaGrid is complementary to the existing GT4. Our experimental evaluation shows that DynaGrid effectively utilizes Grid resources by allocating only the required number of resources adaptively according to the amount of incoming requests, providing both the scalability and the reliability at the same time.

E.-K. Byun
Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST),
373-1 Guseong-dong, Yuseong-gu,
Daejeon 305-701, South Korea
e-mail: ekbyun@camars.kaist.ac.kr

J.-S. Kim (✉)
Sungkyunkwan University,
300 Cheoncheon-dong, Jangan-gu, Suwon,
Gyeonggi-do 440-746, South Korea
e-mail: jinsoo@ca.kaist.ac.kr

Keywords WSRF · SOA · GT4 · Adaptive resource provisioning · Distributed system

1 Introduction

Grid computing is the technology for building Internet-wide computing environment integrating distributed and heterogeneous resources [1]. Grid computing has provided many scientific projects with large computational power and storage capacity. In addition, Grid computing start to provide computational infrastructure for distributed applications based on SOA(Service Oriented Architecture). The recently proposed OGSA (Open Grid Services Architecture) [2] and WSRF (Web Services Resource Framework) [3] specifies the base architecture and interfaces for Grid as the infrastructure of SOA.

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 (or a *ServiceResource*[1]) is a set of data values that persist across, and evolve as a

[1]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.

result of, Web service interactions. The execution model of stateful Web services are explained in Section 3.1.

In this paper, we argue that any Grid system should meet the following requirements to provide enough resources to Grid applications effectively and stably.

– *Adaptability*
  A Grid system is composed of a large number of heterogeneous resources on which diverse applications are executed. Due to the dynamic nature of the Grid, the amount of resources demanded by each application may change over time. Currently, businesses must acquire more processing power and storage than they need simply to cover peak times. For example, Amazon.com and other web retailers must ensure they can handle peak demand in holiday season, while the IRS(Internal Revenue Service) has a peak in demand in April. For both entities, the hardware is under-utilized the rest of the time [12].
  Adaptive resource provisioning is necessary to handle such a situation elegantly. New resources should be allocated adaptively to the application that requires more computing power or storage. If the service request rate decreases, deployed services can be removed in order to save disk space or to reduce management costs. Unfortunately, most of the current WSRF-based hosting environments do not support adaptive resource provisioning since each service needs to be deployed in advance on the statically-partitioned resources.
– *Scalability*
  One of the most important mechanisms for realizing adaptive resource provisioning is dynamic service deployment with which a new service can be deployed on any resource in the Grid without restarting of the container on the target resource. Dynamic service deployment mechanisms would appear to necessitate a service-specific centralized manager which maintains the locations of currently deployed ServiceResources and monitors the status of Grid resources used by the service. Special attention should be paid to design the Grid system since such a centralized manager easily becomes a performance bottleneck and potentially harms the scalability of the system.
– *Reliability*
  In the large-scale Grid system, the availability of resources also tends to vary dynamically as each resource may leave the system or crash unpredictably at any time. In order to achieve reliable services, the state of running services, i.e. ServiceResources, should be kept available even when the current host leaves the system or experiences unstability, overload, or failure.

Globus Toolkit version 4 (GT4) [4] is a representative Grid middleware supporting WSRF in order to execute or access WSRF-compliant Web services in a standard way. On GT4 Web services can exploit heterogeneous Grid environments with platform-independent Web services protocols and Java-based hosting environment. However, since GT4 can't support discussed requirements, another complementary architecture is required.

Therefore, this paper presents DynaGrid, a new framework which offers adaptive, scalable, and reliable resource provisioning for WSRF-compliant applications. We develop a new dynamic service deployment mechanism to support adaptive resource provisioning. To enhance the scalability, our adaptive resource provisioning mechanism does not rely on any centralized manager. DynaGrid achieves this by providing *Client Proxy* on the client side, a modified client stub which transparently redirects the client's request to the actual location of the corresponding ServiceResource. DynaGrid also distributes the cost of management and recovery by partitioning resources allocated for the service. For reliability, every ServiceResource in DynaGrid is replicated to another resource and all the execution requests for the ServiceResource are logged within the replica. In case the original resource crashes, DynaGrid recovers the ServiceResource by replaying logged requests on the replica.

We implemented DynaGrid framework as a set of WSRF-compliant Web services so that they are executed on GT4 container without any modification of GT4 container core. The experimental evaluations are performed on a real testbed with practical applications including the MapReduce application [11]. The results indicate that DynaGrid effectively utilizes Grid resources with scalability and reliability.

The rest of the paper is organized as follows. In Section 2, we briefly overview the related work. Section 3 presents the execution model and the overall architecture of DynaGrid. The proposed mechanisms for adaptive resource provisioning is explained in Section 4. DynaGrid's architecture for scalability and reliability are described in Section 5 and Section 6, respectively. Section 7 presents the evaluation results. Finally, we conclude in Section 8.

2 Related Work

M. Welsh et al. proposed SEDA (staged event-driven architecture) in [13]. SEDA enables Internet services to be concurrently executed on distributed resources and resource allocation and load balancing are automatically handled. Hiding complex resource management from client is similar to DynaGrid. However, SEDA can support only specific application model named *stages* which is a standard as Web service.

Djilali et al. investigated fault-tolerant environment for RPC programming in Internet connected Desktop Grid [14]. They exploit three-tier architecture (client, coordinator, server), message logging, hierarchical fault detection and passive replication. These schemes were well known and DynaGrid also exploits such schemes. DynaGrid especially focuses on keeping ServiceResources alive in WSRF environment under Grid resource failure.

The latest version of GT4 provides remote and dynamic service deployment mechanism called HAND [5]. HAND adds an internal function into the GT4 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 GT4 container.

DynaGrid is the first attempt to build adaptive resource provisioning framework for WSRF-compliant applications with the emphasis on scalability and reliability issues. Our dynamic service deployment mechanism differs from HAND in that our approach requires no modification to the GT4 container. Moreover, ServiceResource replication and recovery are unique to DynaGrid and, to the best of our knowledge, any similar mechanism has not been investigated for WSRF in the previous work.

3 WSRF and DynaGrid Overview

3.1 Web Service Execution Model in WSRF

In this section, prior to explain the architecture of DynaGrid, we will describe how stateful Web services are executed on WSRF environment, how Web services are accessed by clients and eventually how they construct distributed applications. Web services require hosting environment called *Web service container*, for example, Apache Tomcat and IBM Websphere, to be executed on a network accessible host. GT4 container is one of Java-based hosting environments for WSRF-compliant Web services. GT4 container is executed on each host and it allows Web services to utilize the underlying resources of the host such as CPU cycle, memory, and storage. 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. Clients then access the container to execute functions of Web service with SOAP and HTTP.

The most important feature of WSRF compared to traditional Web service is that WSRF defines standard interface for managing the state of resources through Web services. According to the WSRF specification, each client can create its own *ServiceResource* and store the result data of consequent Web service executions in its ServiceResource to keep the client's context on the Web service. All information of Web service execution of client is stored in ServiceResources and they are completely independent from Web service code. A Web service is accessed by client with an endpoint reference containing a URI which points both Web service code and the key representing a specific ServiceResource.

Such separation of Web service code and ServiceResource make it possible for client's context of the Web service to be resumed on any hosting environment in which the corresponding Web service code is deployed and ServiceResource is located. DynaGrid realize ServiceResources to be freely moved to any container for adaptive and reliable execution of Web service.

Such Web services can construct various types of distributed applications. Firstly, Web service itself can be a simple application which handles requests from several clients as in traditional Web servers. Secondly, Web service can be used in a parallel application which performs the same evaluation on diverse input data at the same time. In this case, the core of evaluation is implemented as Web service code and a special management agent creates the necessary number of Service-Resources and execute the Web service for different inputs. The execution may be handled in distributed resources for the performance. Finally, Web services can be components of a multi-tier distributed application. In this case, one Web service takes a role of the client of next tier's Web service. In each tier, plural ServiceResources may be activated concurrently.

3.2 Overall Architecture of DynaGrid

In order to execute a Web service on GT4, Web service have to be deploy on the GT4 container of statically assigned Grid server so that all requests from clients are handled in the pre-assigned container. DynaGrid complements such limitation of the existing WSRF-compliant Grid environment especially GT4. DynaGrid enables Web services to exploit any amount of Grid resource whenever the amount of necessary computational resources varies.

Figure 1 shows the overall architecture of DynaGrid. DynaGrid is composed of four components: ServiceDoor, Dynamic Service Launcher (DSL), PartitionManager, and Client Proxy. In DynaGrid framework, the components of DynaGrid are independent Web services executed on GT4 containers and DSL and Partition-Manager are deployed on every container in advance.

In order to be executed in DynaGrid, every Web service has to deploy a specific Web service named ServiceDoor on a trusted resource and publish its address to clients. The ServiceDoor is the entry point to access the Web service from clients. Only through the ServiceDoor, clients can create new ServiceResources. Clients feel that the ServideDoor is the Web service itself and interact as they access the standard Web service.

ServiceDoor customized for each Web services is automatically created by *DoorCreator* which DynaGrid provides. DoorCreator interprets WSDL (Web Services Description Language), WSDD (Web Services Deployment Descriptors), and the corresponding Web service code.

Beyond ServiceDoor, Web service is actually executed on the GT4 container of the dynamically allocated Grid resource. Dynamic Service Launcher (DSL) is a special Web service running on every resource in DynaGrid. Through DSL, DynaGrid can dynamically deploy new Web services, create ServiceResources, and handle Web service execution request on any GT4 container. DSL also performs ServiceResource replication and request logging.

DynaGrid groups DSLs which are allocated to a specific Web service into several Service-Partitions for the scalability reason. Grouping DSLs is equivalent to grouping GT4 containers since there is only one DSL on a GT4 container. Each ServicePartition consists of at least two DSLs and one PartitionManager. Partition-Manager performs creation, replication, load balancing, and recovery of ServiceResources. In addition, it monitors the status of DSLs and ServiceResources that belong to the same ServiceParition.

ClientProxy is a modified client stub. It transparently redirects the incoming service execution request to the actual location of ServiceResource. DynaGrid also provides a utility called *Proxy-Creator* to automatically generate Client Proxy code from WSDL of the target Web service.

In a service-oriented workflow system, a service can be a client of other service. In such a case, the client-side service should use Web service interface to connect to the server-side service. Therefore in DynaGrid framework, communication between component services of a workflow

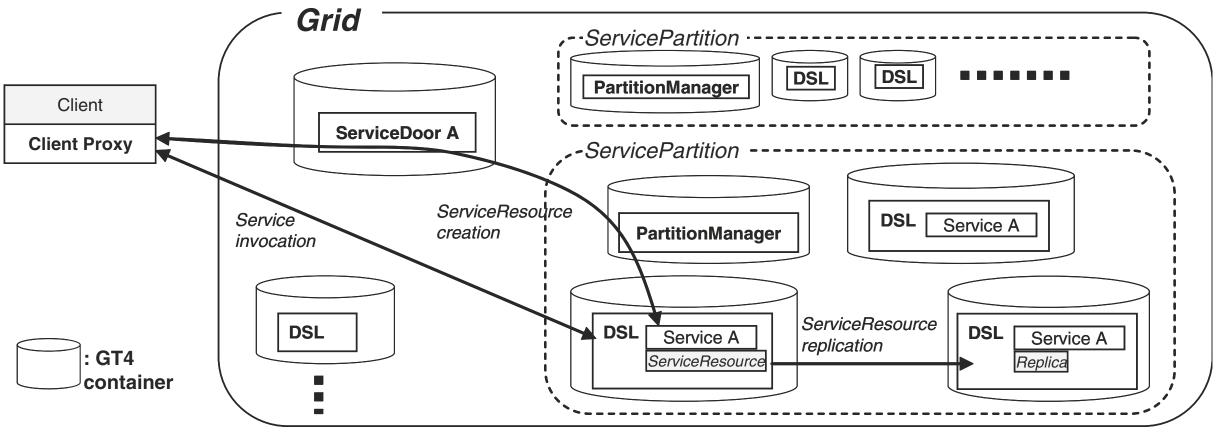

Fig. 1 The overall architecture of DynaGrid

system should pass the ServiceDoor of server-side service.

In the following sections, we describe details of these components and how they interact each other to achieve adaptive, scalable, and reliable resource provisioning.

4 Adaptive Resource Provisioning

The most distinctive feature of DynaGrid is that Web services can be executed on any resource whenever they want. To realize this, we developed a new *dynamic service deployment* mechanism and implemented it as the main function of DSL. Its details will be discussed in Section 4.1.

DynaGrid dynamically allocates additional resources when the current resources can't properly support the increasing demand for Web services and ServiceResources. Resource inadequacies happen in two situations. The first case is when there are too many requests for ServiceResource creation to accommodate them within the allocated resources. The second case is when a ServicePartition has no available resource to replicate ServiceResources and replicas from failed resources (cf. Section 6). In both cases, ServiceDoor dynamically deploys the service into the DSL of a new resource and inserts the DSL to the current ServicePartition.

Because of the additional management step caused by the existence of ServiceDoor, internal steps of ServiceResource creation and Web service execution are modified. Section 4.2 and Section 4.4 will explain the details.

4.1 Dynamic Service Deployment

Dynamic service deployment of DynaGrid is distinct from that of GT4. Typical Web service containers including the GT4 container launch the service code directly when requests to a specific Web service arrive. Each container maintains a list of currently deployed services. Deploying a new service requires adding a new entry into the container's deployed service list. Since the change in the list is usually reflected only after the container is restarted, the latest version of GT4 has a special mechanism which accesses the internal data structure of GT4 in order to reload the deployed service list dynamically [5]. This is why the dynamic service deployment mechanism in GT4 depends on the implementation of the GT4 container. In DynaGrid, on the other hand, deploying a new service does not require any interaction with the underlying Web service container. Every service deployment request is forwarded to an appropriate DSL first. DSL then changes the execution environment to that of the target service and launches the service code. In this way, DynaGrid's dynamic service deployment mechanism conforms to WSRF specifications and thus can be used in any WSRF-compliant hosting environments.

In order for DSL to manage dynamically deployed services, a new type of ServiceResource named *Meta ServiceResource* is introduced. Once a new service is deployed, a new Meta ServiceResource is created. It stores the information on the deployed service including the service ID, interface class, service options, and ClassLoader. With the endpoint reference (EPR) of the Meta ServiceResource, other components in DynaGrid can decide the target Web service for ServiceResource creations and service executions.

Typical steps of dynamic service deployment in DynaGrid are as follows. (1) ServiceDoor searches for an available resource using the information service of the Grid. (2) It transfers the package file containing the service code and ServiceResource class to the resource, and deploys the service through DSL on the new resource. (3) DSL returns the EPR of the created Meta ServiceResource. (4) Finally, the returned EPR is inserted into a ServicePartition and the PartitionManager begins to monitor the status of the DSL.

4.2 ServiceResource Creation

The first step to invoke a Web service is to create a ServiceResource and to obtain its EPR composed of the key and the URI of the service. Clients ask ServiceDoor to create a ServiceResource through Web service interfaces. Addresses of all the created ServiceResources are kept by ServiceDoor so that clients can inquire the actual locations of ServiceResources to ServiceDoor.

Figure 2 depicts typical steps to create a new ServiceResource in DynaGrid. When ServiceDoor receives the creation request, it assigns a *global key* and select in which ServicePartition new ServiceResource will be created. ServiceDoor periodically gather state information from every PartitionManagers to check how many rooms each ServicePartition have to create new ServiceResource. Based on that information, ServiceDoor relays the creation request to the PartitionManager of the most sufficient ServicePartition.

PartitionManager then selects in which container the new ServiceResource will be created. Since PartitionManager also keeps the load information of each resources including CPU utilization, amount of free memory and the number of concurrent processes, it can easily select most under-loaded resource. ServiceResource is created through DSL in the selected resource and the *local EPR* which specifies the actual location of the newly created ServiceResource is returned to ServiceDoor. ServiceDoor then inserts a global key-to-local EPR mapping entry into the mapping table. Note that the local EPR cannot be delivered to clients directly because DynaGrid may dynamically relocate the ServiceResource if necessary. Instead, ServiceDoor returns a new EPR composed of the address of ServiceDoor and the global key as a creation result to the client.

4.3 Web Service Execution

In general, clients use client stubs to invoke Web services. A client stub translates Web service invocation requests from the client to SOAP messages and sends them to remote Web service engines (for example, the GT4 container). Since

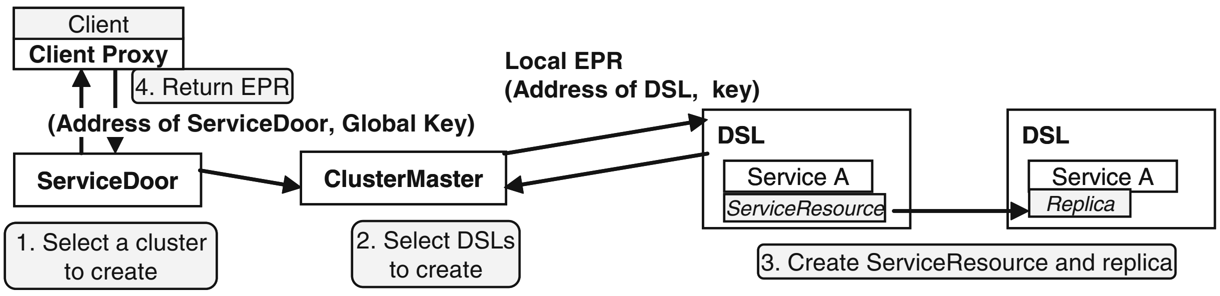
Client
Client Proxy
4. Return EPR
(Address of ServiceDoor, Global Key)
ServiceDoor
1. Select a cluster to create
ClusterMaster
2. Select DSLs to create
Local EPR (Address of DSL, key)
DSL
Service A
ServiceResource
DSL
Service A
Replica
3. Create ServiceResource and replica

Fig. 2 Steps for ServiceResource creation

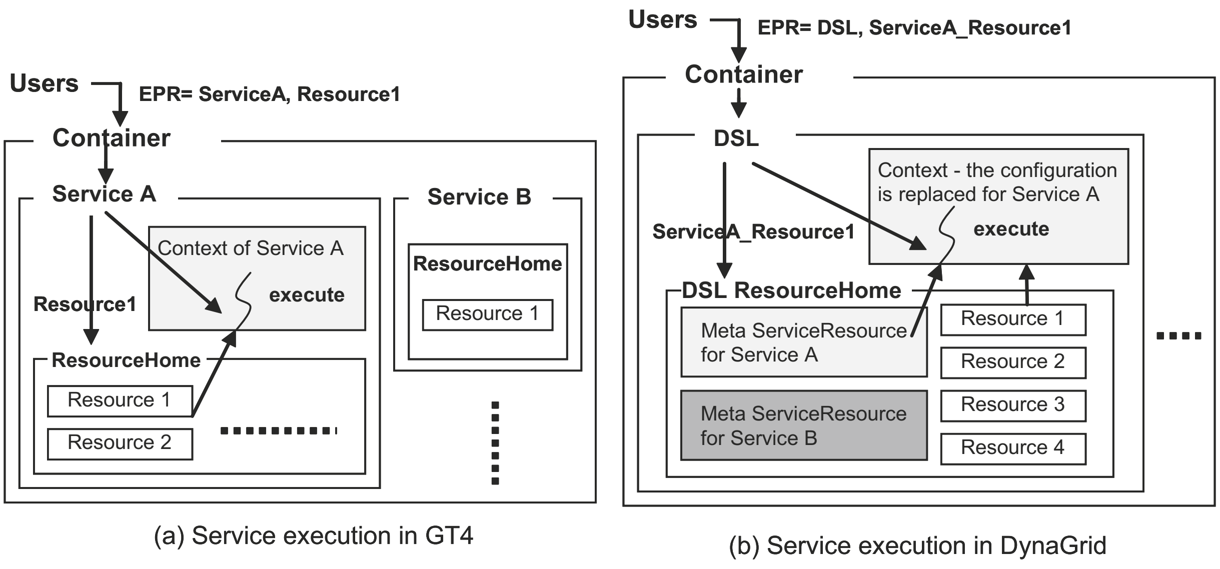

Fig. 3 The comparison of service execution mechanisms in GT4 and DynaGrid

DynaGrid has an enhanced service deployment mechanism, Web service execution procedure is slightly different compared to the traditional Web service engines.

Figure 3a shows how a service is executed in the GT4 container. When a container receives a request from client, it parses the EPR composed of the URI of the service and the ServiceResource key. Then, it 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.

On the other hand, the service execution in DSL proceeds as shown in Fig. 3b. When a container receives a request, it starts a context to execute 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 results to the client.

4.4 Discussion About Independent DSL

DLS can be said a service container on a host service container and it seems inefficient. Frankly, DSL can be implemented as a independent host container such as GT4 container of Apathe Tomcat instead of Web service. In such approach DSL's core functionalities such as dynamic service deployment and ServiceResource can be implemented more efficiently without using unnecessarily complex mechanisms such as MetaServiceResource and ClassLoader relocation. The advantage of Web service form of DSL is that DynaGrid can be easily applicable to VO constructed with GT4 containers because deploying new Web service is much easier than deploying new container to all hosts. The contribution of Web service implementation of DSL is that it can complement the functionalities required for DynaGrid without any modification of GT4 container.

5 Scalability in DynaGrid

Although ServiceDoor and dynamic service deployment mechanism successfully enable adaptive resource provisioning, the presence of the centralized ServiceDoor causes a long service execution path and the possibility of being a bottleneck. In this section, we describe two clever designs to improve the scalability of DynaGrid. First, distributed PartitionManagers deal with most of complex management tasks such as creation, replication, load balancing, and recovery on behalf of ServiceDoor. Second, ClientProxy allows to bypass ServiceDoor during Web services invocations. The following subsections elaborate upon the features provided by PartitionManager and ClientProxy.

5.1 ServicePartition

In DynaGrid, there must be a central manager to handle ServiceResource creation and to continuously monitor the status of resources for adaptive resource provisioning. If all these tasks are performed in ServiceDoor, ServiceDoor can be easily overloaded and DynaGrid cannot be scalable as the demand for resources increases. To prevent ServiceDoor from being overburdened, DynaGrid delegates some management tasks, such as ServiceResource creation, replication, recovery, and load balancing, to each ServicePartition. ServicePartition is composed of a PartitionManager and DSLs on which Web services are deployed by the corresponding ServiceDoor. Each PartitionManager independently manages its own DSLs and ServiceResources.

When a PartitionManager receives the ServiceResource creation request from ServiceDoor, it first selects two most under-utilized DSLs in the ServicePartition; one is for the original and the other is for its replica. After successful creation of the new ServiceResource and its replica on the selected DSLs, the PartitionManager starts to keep track of the local EPR and the replica location of the new ServiceResource. The local EPR of the newly created ServiceResource is returned to the ServiceDoor.

PartitionManager periodically monitors DSLs to see whether the resource is overloaded or to collect the list of destroyed ServiceResources. Those data are used to choose the best DSL when PartitionManager tries to create new ServiceResources or replicas. When it detects the failure of DSL, the recovery mechanism is invoked (cf. Section 6.2).

Along with the management of ServicePartition, PartitionManager also reports its status to ServiceDoor periodically. In order to prevent PartitionManager from being a performance bottleneck, the size of a ServicePartition is limited. When a ServicePartition is too overloaded, ServiceDoor dynamically creates a new ServicePartition to handle the increased service requests. Through this hierarchical resource management structure, we can improve the scalability of DynaGrid.

5.2 Direct Web Service Invocation

In DynaGrid, ClientProxy generated by ProxyCreator replaces the client stub. Every Web service client should load Client Proxy instance for a specific ServiceResource. ClientProxy transparently redirects the request to the DSL where the ServiceResource actually exists. Without ClientProxy, every service execution request should pass through ServiceDoor, which will limit the scalability of DynaGrid.

Figure 4 depicts steps for Web service invocation in DynaGrid. Every EPR known to clients is composed of the address of the ServiceDoor and a

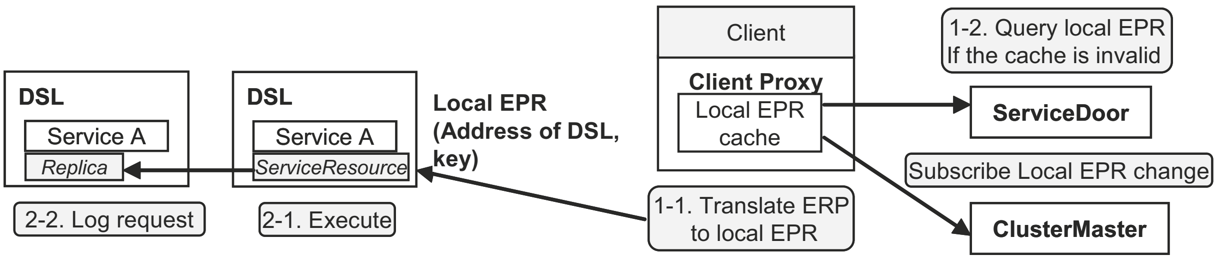
Client
1-2. Query local EPR If the cache is invalid
DSL
DSL
Local EPR (Address of DSL, key)
Client Proxy
Local EPR cache
ServiceDoor
Service A
Replica
Service A
ServiceResource
Subscribe Local EPR change
2-2. Log request
2-1. Execute
1-1. Translate ERP to local EPR
ClusterMaster

Fig. 4 Steps for Web service invocation

global key as described in Section 4.2. On the first access, ClientProxy obtains the local EPR according to the global key from ServiceDoor and caches the returned EPR internally. Thereafter, successive execution requests are delivered directly to the ServiceResource bypassing ServiceDoor.

Since the location of the ServiceResource may change, ClientProxy also obtains the EPR of the PartitionManager from ServiceDoor and subscribes to the PartitionManager to receive the notification when the local EPR changes. If the cached local EPR becomes invalid due to the failure of the corresponding DSL, the up-to-date EPR is immediately fetched from ServiceDoor.

6 Reliability in DynaGrid

The failure of a resource generally leads to total loss of ServiceResources served on the resource. If we know that a ServiceResource will be unaccessible, we can migrate it to another stable resource for providing uninterrupted service. However, all of the existing WSRF-compliant systems are designed in such a way that each ServiceResource is handled only on its birthplace since WSRF specifications do not define any feature related to ServiceResource relocation. DynaGrid resolves this problem with the recovery mechanism coupled with ServiceResource replication and request logging.

We assume that ServiceDoor is executed on a reliable host. Thus, DynaGrid provides no recovery mechanism for ServiceDoor at this moment. This assumption is acceptable since ServiceDoor is designed not to be overloaded and it occupies only one resource even if there are huge demands for the service.

6.1 ServiceResource Replication

DynaGrid provides a mechanism for ServiceResource replication. Every ServiceResource has its own replica on another DSL in the same ServicePartition. When a ServiceResource is created, PartitionManager makes its replica on another DSL by storing the first snapshot of the ServiceResource. All the following execution requests for the ServiceResource are logged within the replica. By replaying logged requests on the snapshot, ServiceResource can be recovered from its replica. The new snapshot of a ServiceResource is replicated whenever the specified number of requests are logged. Eventually, each replica stores the most recent snapshot and logs of requests issued after the last snapshot. This policy can reduce the time for replaying logged requests and save the storage for logs. Making snapshot is also performed if the execution request cause some side-effect so that reply may break the consistency. We add a field to the operation description in WSDL to show whether the operation cause side-effect or not. Unfortunately application developers should check the field manually in the current version of DynaGrid.

DSL provides a Web service method for transferring serialized ServiceResource object and storing it in the corresponding replica. Before making a snapshot of a ServiceResource, DSL acquires the lock on the ServiceResource to make sure the ServiceResource is not being modified by any execution request. DynaGrid also provides a group of APIs which help service developers to easily implement their ServiceResources as Serializable objects.

A request logging is processed in the background during the request is executed. The result is returned to the client only after both the logging and the service execution finish successfully. Note that a request logging may take more time than the request execution since it requires one more Web service invocation to the replica. Moreover, if the DSL of the replica crashes, the logging would be delayed further for the recovery of the replica. In spite of such possibility of long delay, our synchronous logging scheme is essential for the reliability. Without synchronized logging, the request log may be permanently lost since the client may not be aware of the loss of the replica, failing to re-issue the lost request. In this case, ServiceResources can not be properly recovered. In brief, all Web service requests from clients are executed atomically.

We selected the number of replicas as just one to minimize the overhead including network traffic and storage. Section 7.4 verifies that the single replica policy properly handles the reliability of Web service. Nonetheless, DynaGrid can be extended to manage multiple replicas to handle the Grid environment composed of very volatile resources and networks. In such cases, each Web service execution may takes more time because it waits until the log is stored in all replicas.

6.2 Recovery of DSL

In DynaGrid, the information on a ServiceResource is distributed among DSL, PartitionManager, and ServiceDoor. DSL has all the information on the original ServiceResources as well as the snapshots and request logs of the replicas. PartitionManager keeps a mapping table between the ServiceResource key and the addresses of DSLs where the ServiceResources and their replicas reside. ServiceDoor maintains the local EPR of every ServiceResource.

The failure of a DSL can be detected in two ways. First, since every DSL periodically notifies its status to PartitionManager, PartitionManager can notice that a DSL has failed if the notification is not arrived for the configured time. In the current DynaGrid, notifications are sent every 6 s and the timeout value is 20 s. Second, the failure of a DSL can be detected by ClientProxy when it accesses the ServiceResource or by other DSLs when they send request logs to the DSL. As soon as Client Proxies or other DSLs see errors, they ask PartitionManager to check the availability of the ServiceResource or the replica. If PartitionManager observes the DSL failure, it starts to recover all the ServiceResources and replicas based on its mapping table. Some errors can be caused by the incorrect location information in Client Proxies or DSLs, in which case PartitionManager returns the correct information and lets them retry.

If a DSL fails, PartitionManager recovers all the ServiceResources and replicas simultaneously. Figure 5a illustrates the recovery steps for DSL failure. To recover a replica, PartitionManager creates a new replica on another DSL (step 2–2) and immediately replicates the current snapshot of the ServiceResource (step 3). PartitionManager updates its ServiceResource mapping table (step 4) and informs the original DSL that the location of the replica is changed. The recovery process for a ServiceResource is similar, but PartitionManager first needs to order the DSL of the replica to replay logged requests to make the up-to-date ServiceResource (step 2–1). It then creates a new replica (step 2–2) and replicates the snapshot of the ServiceResource from the replica

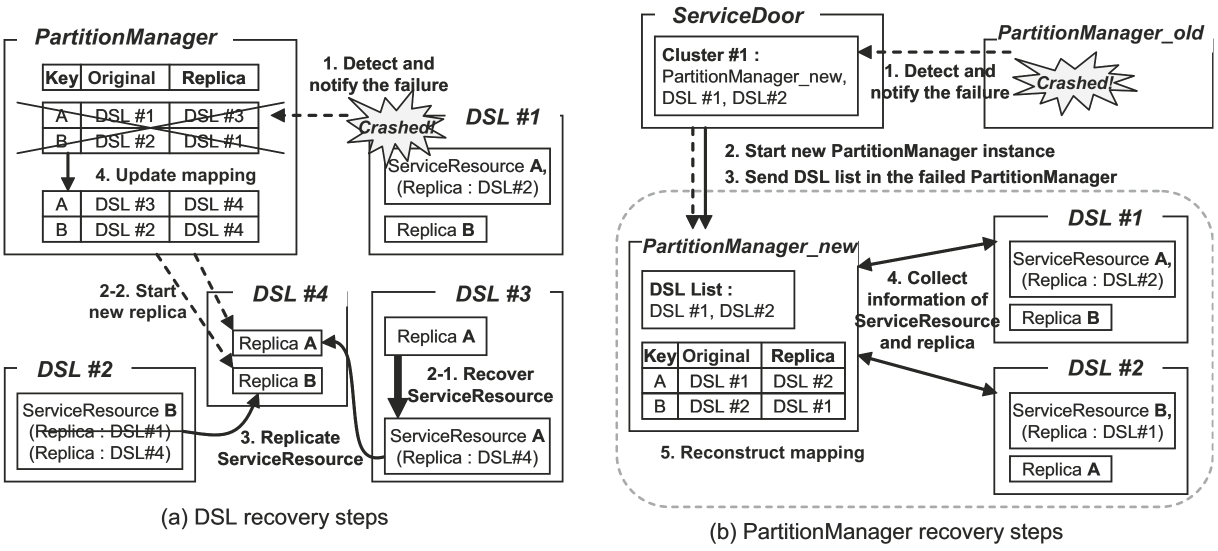
PartitionManager
Key Original Replica
A DSL #1 DSL #3
B DSL #2 DSL #1
4. Update mapping
A DSL #3 DSL #4
B DSL #2 DSL #4
1. Detect and notify the failure
Crashed!
DSL #1
ServiceResource A, (Replica : DSL#2)
Replica B
2-2. Start new replica
DSL #4
Replica A
Replica B
DSL #3
Replica A
2-1. Recover ServiceResource
ServiceResource A (Replica : DSL#4)
DSL #2
ServiceResource B (Replica : DSL#1) (Replica : DSL#4)
3. Replicate ServiceResource
(a) DSL recovery steps
ServiceDoor
Cluster #1 : PartitionManager_new, DSL #1, DSL#2
PartitionManager_old
1. Detect and notify the failure
Crashed!
2. Start new PartitionManager instance
3. Send DSL list in the failed PartitionManager
PartitionManager_new
DSL List : DSL #1, DSL#2
Key Original Replica
A DSL #1 DSL #2
B DSL #2 DSL #1
4. Collect information of ServiceResource and replica
5. Reconstruct mapping
DSL #1
ServiceResource A, (Replica : DSL#2)
Replica B
DSL #2
ServiceResource B, (Replica : DSL#1)
Replica A
(b) PartitionManager recovery steps

Fig. 5 The recovery process in DynaGrid

(step 3). Finally, the mapping table is updated (step 4).

PartitionManager can misjudge the failure of DSL due to the network problem. Moreover, DSL looks alive from client while it looks failed from PartitionManager or ServiceDoor. In such cases, requests already issued to the misjudged DSL are properly finished and next requests are executed on new DSL since PartitionManager notifies the address of new DSL to ClientProxy of the client. The misjudged DSL is removed from the allocated DSL list of PartitionManager. After the network problem is resolved, the DSL may be allocated to the same Web service.

Client are also able to crash. If some clients become unavailable, their ServiceResource remains uncontrolled. These orphan ServiceResources are swept according to the WS-Lifetime, one of the WSRF standard specification. Every ServiceResource has its own lifetime. In DynaGrid, DSLs and PartitionManagers automatically remove old ServiceResources and its replica and all logged requests.

6.3 Recovery of PartitionManager

ServiceDoor can detect the failure of PartitionManager when the notification from PartitionManager times out. ServiceDoor can also sense the failure if a creation request on the PartitionManager throws an error.

PartitionManager has a list of ServiceResources in the ServicePartition and the mapping table which maintains the locations of ServiceResources and their replicas based on the ServiceResource keys. This information is duplicated in ServiceDoor and DSLs so that DynaGrid can reconstruct them even though a PartitionManager fails. Recall that ServiceDoor knows the list of DSLs for all ServicePartitions and each DSL keeps the locations of replicas of all the ServiceResources it has.

The recovery steps for PartitionManager failure are depicted in Fig. 5b. When ServiceDoor observes the failure of PartitionManager (step 1), ServiceDoor spawns a new PartitionManager (step 2) and sends the list of DSLs which were managed by the failed PartitionManager (step 3). The new PartitionManager contacts all the DSLs to gather the list of ServiceResources and replicas (step 4). Based on the collected information, the new PartitionManager rebuilds the mapping table (step 5). After the mapping table is rebuilt, PartitionManager searches for ServiceResources and replicas that were failed during the recovery of PartitionManager, and recovers them if any. Finally, it starts to manage the ServicePartition.

6.4 Load Balancing

Although PartitionManager always tries to dispatch ServiceResource creation requests to lightly-loaded DSLs, the imbalance in resource utilization between DSLs is unavoidable since the usage pattern of ServiceResources changes over time. To cope with such a situation, DynaGrid provides a load balancing mechanism using ServiceResource replication.

All DSLs in each ServicePartition periodically notify their resource status to their PartitionManager. If the status violates the configured load balancing threshold, the PartitionManager starts a load balancing mechanism. In the current implementation of DynaGrid, the service provider can set the various load balancing threshold based on CPU utilization, the occupied memory size, and the number of concurrent ServiceResources serviced per DSL.

If the load balancing mechanism is activated, PartitionManager selects one third of ServiceResources in the busy DSL. The ratio 'one third' is selected to balance between the overhead of ServiceResource migration and the obtainable room in the busy DSL. Of course, the optimal ratio can vary according to the resource demand change of the Web service. The current selection criterion is the load of the DSL on which the replica is located. In other words, ServiceResources whose replicas reside in lightly-loaded DSLs are selected. Once a ServiceResource is selected for load balancing, it is moved from the overloaded DSL to another DSL in the same way as the ServiceResource is failed. If the ServiceResource cannot be accommodated in the existing DSLs, PartitionManager asks ServiceDoor to allocate additional DSLs.

7 Evaluation

7.1 Environment and Benchmarks

We construct a testbed with 22 servers connected by 100Mbps Ethernet. Table 1 summarizes the list of servers used for our testbed. On each server, Linux 2.4.20 and JDK 1.4.2 are installed and the container of Globus Toolkit version 4.0.1 is used as the hosting environment.

For our evaluation, we implemented three practical applications as WSRF-compliant Web services. *Ray tracing service* is a Web service implementation of RAJA [7], an open source ray tracer. Its ServiceResource maintains a result buffer and a request queue on which scene data and the required quality operations are placed. *Streaming buffer service* exploits the server's physical memory to buffer streaming data similar to RBNB (Ring Buffered Network Bus) [8]. RBNB has been used in many Grid systems including NEESGrid [9] for data aggregation, streaming, and synchronization. The ServiceResource of streaming buffer service includes the buffered data received from a source channel. Finally, in order to evaluate the overall effectiveness of DynaGrid, we developed a MapReduce [11] system called *GridMR* on DynaGrid and executed Apache Nutch [10], an open source search engine written in Java, on GridMR.

As a microbenchmark, we also implemented *Add* Web service, which simply performs add operation and keeps the result of the calculation as ServiceResource. The Add Web service is used to analyze the overheads of DynaGrid.

7.2 Adaptive Resource Provisioning

We carried out two experiments to show that DynaGrid adaptively manages its resources even when the request rate from clients sharply increases. In these experiments, DSLs are deployed only on Pentium-4 servers in our testbed.

Table 1 List of servers in our testbed

Count	CPU	RAM	SMP
8	Pentium-4 2.8 GHz	1 GB	2-Way
6	Pentium-4 2.8 GHz	512 MB	No
8	Pentium-3 850 MHz	1.5 GB	2-Way

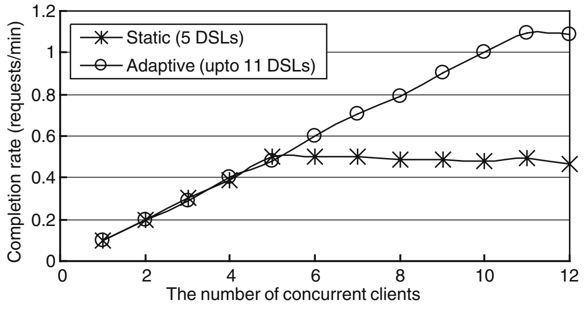

Fig. 6 The comparison of the request completion rate in the ray tracing service

First, we compare the request completion rate in the ray tracing service when the number of clients increases from 1 to 12 as shown in Fig. 6. We use total 11 DSLs in this experiment. For static provisioning, we allocate only five DSLs assuming the initial administrative decision suggests that five resources are enough. For adaptive provisioning, we allow DynaGrid to use all the DSLs and each DSL is limited to serve only one tracing request at a time so that the superfluous requests can be directed to other DSLs. Each client repeats the same request which takes about 10 min. After the number of clients reaches five in Fig. 6, static provisioning does not appear resilient enough to support the unexpected resource demand. However, adaptive provisioning makes use of all the resources, effectively handling the increased request rate.

Second, we measure the amount of occupied memory by the streaming buffer service. Figure 7 depicts the change in the aggregated buffer size

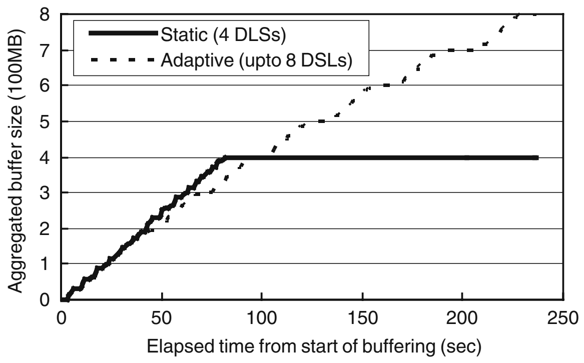

Fig. 7 The comparison of the aggregated buffer size in the streaming buffer service

of 24 streaming channels on which data are generated at 200 KB/s from sources. In the case of adaptive provisioning, the service is initially deployed on two DSLs and all channels are created on them. On each DSL, the total memory capacity allocated to buffers is limited to 100 MB. When the total size of buffers reaches 100MB in each DSL, the load balancing mechanism of DynaGrid is triggered to relocate one third of buffers. DynaGrid may allocate additional DSLs (up to eight DSLs) to the streaming buffer service to obtain more memory. In the static provisioning case, the channels are evenly distributed over four DSLs. Figure 7 presents that, with the adaptive provisioning scheme, the aggregated buffer size is constantly expanded until the streaming buffer service fully occupies the configured memory of every DSL. Meanwhile, the total buffer size is limited to 400 MB in the static provisioning case. Adaptive provisioning shows the slightly slower growth rate because the streaming buffer service is temporarily blocked during the relocation of ServiceResources.

Although DynaGrid enables flexible resource management, it introduces extra overhead compared to normal Web services. Table 2 presents the overhead of dynamic service deployment measured during the experiment shown in Fig. 6. Initializing a ServicePartition takes 1,009 ms where starting PartitionManager instance and adding two DSLs into the ServicePartition require 530 and 479 ms, respectively. We can observe that most of the time during dynamic service deployment is spent to transfer the service code, whose size is 1,890 KB for the ray tracing service. Even if the overhead seems to be high, this overhead can be amortized considering that dynamic service deployment does not happen frequently. Two previous experiments confirms this as the overall throughput is not seriously affected by dynamic service deployment.

Table 2 Overhead of dynamic service deployment in the ray tracing service

Operation	Time (ms)
Initializing a ServicePartition	1,009
Starting PartitionManager instance	530
Adding two DSLs into ServicePartition	479
Dynamic service deployment	2,050
Transferring service code (1,890 KB)	1,699

7.3 Scalability

7.3.1 Overhead of Service Execution in DynaGrid

Table 3 summarizes the ServiceResource creation and execution time compared to the normal Web service deployed on the GT4 container. The ServiceResource creation time on DynaGrid requires 654 ms on average which is almost five times longer than the direct creation. This overhead is mainly caused by the additional network steps required for contacting ServiceDoor and PartitionManager, creating a replica, and transferring the first snapshot. Again, this overhead can be amortized since the ServiceResource creation occurs only once.

A Web service invocation takes 174 ms including the request logging in the replica. Note that, without ServiceResource replication, DynaGrid shows the same performance as direct execution due to ClientProxy. If ClientProxy has an invalid local EPR, DynaGrid requires additional 104 ms to query ServiceDoor. The logging overhead of 70 ms can be hidden for time-consuming Web services since the logging is a simple Web service call to the replica DSL and it is performed concurrently with the actual Web service execution.

7.3.2 Improvement by ServicePartition

This experiment is to show the effectiveness of PartitionManager in the ServiceResource

Table 3 Overhead of ServiceResource creation and execution on DynaGrid

Operation	Time (ms)	Ratio
Direct creation on GT4	130	1
Creation in DynaGrid	654	5.03
ServiceDoor/P.Manager latency	193	1.48
Replica creation	127	0.97
Copying the first snapshot	204	1.57
ServiceResource creation	130	1
Direct execution on GT4	105	1
Execution in DynaGrid	174	1.65
Request logging	70	0.67
Via ServiceDoor	275	2.62
Query ServiceDoor	104	0.99

creation. ServiceDoor and PartitionManagers run on Pentium-4 servers, and DSLs on eight Pentium-III servers. We compare the ServiceResource creation throughput using one PartitionManager and four PartitionManagers. Figure 8 shows that four PartitionManagers achieve the better throughput than a single PartitionManager. The result is because, even though all ServiceResource creation requests should pass ServiceDoor inevitably, PartitionManager takes most part of the creation procedure and these are distributed among four PartitionManagers. The figure also shows the maximum throughput of ServiceDoor for Resource creation. Assuming there is no overhead spent in PartitionManager and DSL during creation, in our testbed 15 ServiceResources can be created in each second.

ServicePartition also makes DynaGrid scalable by sharing the monitoring overhead. PartitionManager constantly collects monitoring messages from DSLs in the ServicePartition to check the status and the availability of each DSL. Assuming that ServiceDoor directly monitors all the DSLs, ServiceDoor can be easily overloaded by a number of DSLs. In Fig. 9, we measure the average CPU load of PartitionManager caused by handling notification messages from DSLs. As shown in Fig. 9, the monitoring overhead is not negligible when there are hundreds of DSLs and beyond. Using several ServicePartitions, however, DynaGrid avoids a situation that one ServicePartition becomes overloaded by too many DSLs.

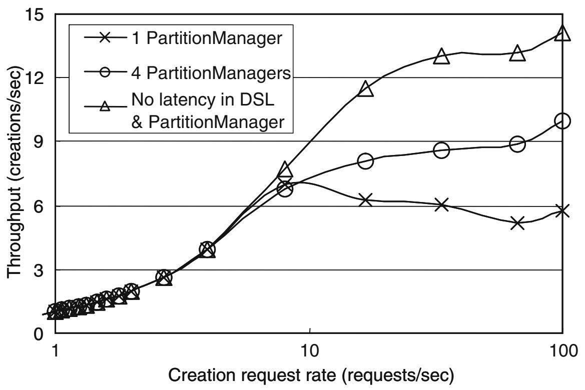

Fig. 8 The comparison of the ServiceResource creation throughput

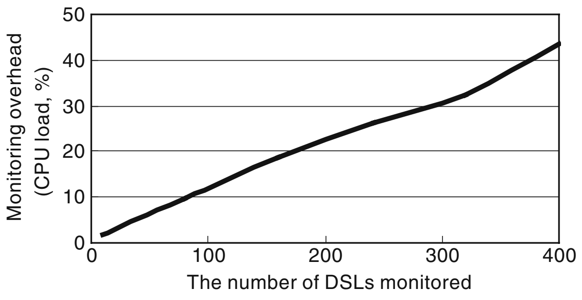

Fig. 9 The overhead of monitoring according to the number of monitored DSLs (each DSL sends a notification every 5 s)

7.3.3 Improvement by ClientProxy

As described in Section 5.2, ClientProxy caches the local EPR of ServiceResource and sends the requests to the ServiceResource directly. In this subsection, we measure the scalability improvement by the direct service invocation quantitatively.

Figure 10 illustrates the throughput achieved with the simple Add service in various set of worker pool composed of Pentium-4 and Pentium-3 servers. We use Pentium-4 servers for ServiceDoor and PartitionManager. In the first three cases, every request from client passes through ServiceDoor. And the rest of the cases use the direct Web service invocation using ClientProxy.

The result presents that, without direct Web service invocation, DynaGrid can not use the

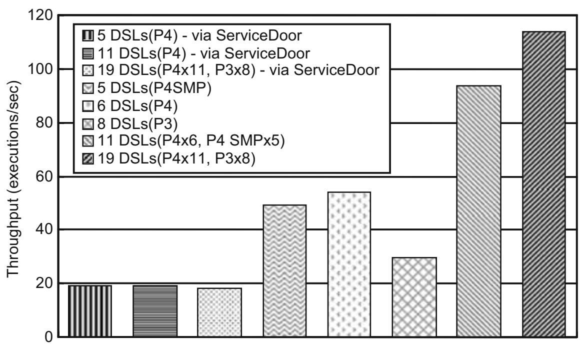

Fig. 10 The effect of ClientProxy for the scalability through the comparison of aggregated throughput

whole resources effectively since ServiceDoor becomes a bottleneck. Even when DynaGrid uses the same number of DSLs, we can see much higher throughput using the direct Web service invocation technique. The result also shows that the throughput with ClientProxy is scalable according to the number of DSLs hosting the service.

7.4 Reliability

Table 4 shows the recovery costs due to the failure of PartitionManager or DSL under various conditions. The time to recover a DSL which has two ServiceResources and two replicas is only 358 ms. For a ServicePartition which hosts 36 ServiceResources and 36 replicas on 18 DSLs, PartitionManager can be recovered from the failure in 2,039 ms. From Table 4, we can see that most recovery can be completed in a couple of seconds.

DSLs may remain unrecovered until PartitionManager detects the failure of DSL or until ServiceDoor detects the failure of PartitionManager. In the current configuration, ServiceDoor and PartitionManager check the availability of PartitionManager and DSLs, respectively, every 20 s. In case both PartitionManager and DSLs in the same ServicePartition fail at the same time, few more seconds are necessary for the recovery of PartitionManager. Therefore, the time a ServiceResource remains down can be more than 20 s in the worst case.

Latencies experienced by client due to the recovery of nodes keeping ServiceResource and replica are 1085 ms and 631 ms respectively. These latencies contain the time for checking the failure through PartitionManager and the delay caused by the recovery process. Since ServiceResource recovery process contains checkpointing and replication of ServiceResource snapshot, the recovery time mainly depends on the size of ServiceResource.

Table 4 Summary of recovery costs (SRs: ServiceResources, reps: replicas)

Target	Size	Time (ms)
DSL	2SRs, 2reps	358
	16SRs, 16reps	1,886
	20SRs, 20reps	2,474
PartitionManager	4DSLs, 4SRs, 4reps	657
	4DSLs, 40SRs, 40reps	825
	18DSLs, 36SRs, 36reps	2,039

Generally the size of ServiceResource hardly exceeds tens of megabytes and consequently recovery time is smaller than 10 s. Moreover, the failures do not happen frequently, the overhead is acceptably small compared to the overall execution time.

The survival duration of resources participating in the Grid can be inferred from the previous study on peer-to-peer systems. From the observations on Kazaa and Gnutella, Saroiu et al. found that the average session duration in peer-to-peer system was about 60 min [6]. Notice that, for the Grid system composed of under-utilized resources of universities or research institutions, the average survival duration would be much longer. Moreover, if a resource leaves the Grid system not by the accident but by intention, the resource can ask its PartitionManager to relocate ServiceResources safely. Therefore, we can say that 60 min for the average survival duration is a very cautious value in the Grid environment.

In DynaGrid, a ServiceResource is lost when the DSL of its replica crashes during the recovery of the original ServiceResource, and vice versa. Assume that the recovery of a ServiceResource takes at most 30 s and the failure event follows the poisson distribution with the average survival time of 60 min as discussed in the previous paragraph. Under these assumptions, the probability for loss of ServiceResource is estimated to only 0.8%. Remember that those parameter values represent a very unstable environment. If the average lifetime of resource is extended to 12 hours, the ServiceResource loss ratio is lowered to 0.07% and DynaGrid can provide 99.93% of the ServiceResource availability.

7.5 MapReduce on DynaGrid

We select Apache Nutch [10] to evaluate the effectiveness of DynaGrid in practical areas. Nutch generates indexes of web sites crawled from input URLs. The pseudo-code for Nutch is as follows.

```
Inject initial URLs into fetch list;
loop for depths
    Filter fetch list to remove recently
      fetched web pages;
    Fetch web pages in fetch list;
Update fetch list with outgoing links in
  fetched web pages;
end loop;
Generate indexes from fetched web pages;
```

The latest version of Nutch exploits *MapReduce* [11] proposed by Google Inc. MapReduce is a programming model for processing large data sets in distributed environment. In MapReduce, a set of key/value pairs are generated from another set of key/value pairs in two step data processing: Map and Reduce. Each Map and Reduce function can be executed on distributed servers in parallel.

We develop GridMR, a distributed application constructed with WSRF-based Web services which implements the Map and Reduce function and a GridMR library which is a central manager implemented as attachable Java-library. GridMR library created the necessary number of ServiceResources named MRWorkers according to the size of input data and start the execution. Each MRWorker represent intermediate data of a Map or Reduce function. Eventually a GridMR library and several MRWorkers construct a distinctive MapReduce application. Nutch then can inject the input into or obtain the output from GridMR through GridMR library.

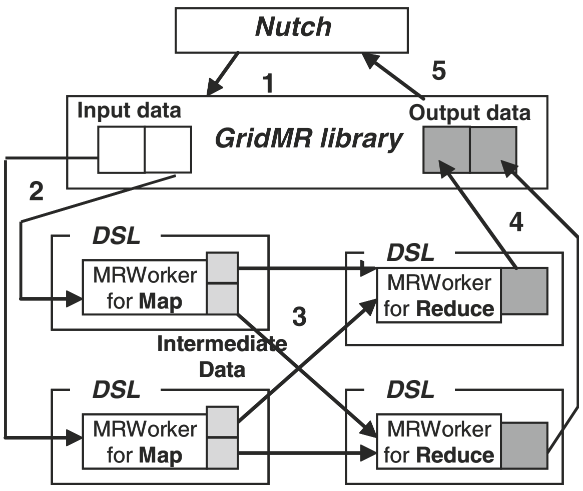

Fig. 11 The execution of GridMR on DynaGrid

Table 5 The execution time (seconds) of Nutch with GridMR on DynaGrid

Environment	Input 1	Input 2	Input 3
Original (Single server)	591	1,435	2,862
GridMR (5 DSLs)	630	1,386	2,497
GridMR (11 DSLs)	383	997	1,832
GridMR (11 DSLs,1 failure)	407	1,059	1,906

Figure 11 describes the detail execution step of GridMR on DynaGrid. Receiving the input data from Nutch (step 1), GridMR library splits the input data and transfers them to a set of MRWorkers in charge of Map function (step 2). They then execute Map function for the input data and transfer the result, called the *intermediate data* [11], to MRWorkers responsible for Reduce function (step 3). After all Reduce MRWorkers finish data processing, GridMR library collects the output data from them (step 4) and returns to Nutch (step 6).

In Table 5, we measured the execution times of Nutch with GridMR for three different inputs and compared them to the performance of the original Nutch on a single server. For GridMR, the number of MRWorkers for Map or Reduce function is configured to be equal to the number of DSLs. Table 5 reports that the performance of Nutch improves as the number of DSLs increases. However, the performance benefit of GridMR is not significant compared to the original Nutch. This is because GridMR uses the time-consuming Web services protocols and the large amount of data are transferred over the network. Notice that GridMR successfully finishes the data processing only with the negligible performance degradation even though a failure occurs in one of DSLs.

8 Conclusions

In this paper, we propose DynaGrid, a new framework which offers adaptive, scalable, and reliable resource provisioning for WSRF-compliant applications. Adaptive resource provisioning is realized by our new dynamic service deployment mechanism and ServiceDoor. ClientProxy and distributed PartitionManagers enhance the scalability of DynaGrid. ServiceResource replication and recovery improve the reliability of DynaGrid. DynaGrid is complementary to the existing Grid middleware such as GT4, and can be used with any Java-based WSRF-compliant hosting environments.

Currently, DynaGrid doesn't provide any additional security mechanism. We assumed that all containers in the same VO are trusted each other and that only certified users can access the host container by other security mechanism in the other layer. We only focused on adaptability, scalability and reliability of WSRF services and ServiceResources. Security issues will be treated in the future version of DynaGrid.

In the current version of DynaGrid, ServiceDoor is assumed to reside in a reliable node. However, practically centralized ServiceDoor can affect the stability of Web service. As future work, we will enhance the DynaGrid to be resilient against the ServiceDoor failure. For example, duplicating ServiceDoor in one or more back-up nodes can be a solution. We plan to release the source code of DynaGrid into the public domain in the near future.

Acknowledgement This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R01-2007-000-11832-0).

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: enabling scalable virtual organizations. Lecture Notes in Computer Science 2150. Springer-Verlag (2001)
2. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed. Globus Project, www.globus.org/research/papers/ogsa.pdf (2002)
3. WSRF: OASIS, Web Services Resource Framework (WSRF) TC, http://www.oasis-open.org/committees/wsrf/
4. Foster, I.: Globus toolkit version 4: software for service-oriented systems. Lecture Notes in Computer Science, vol. 3779, pp. 2–13. Springer-Verlag (2005)
5. Qi, L., Jin, H., Foster, I., Gawor, J.: HAND: highly available dynamic deployment infrastructure for globus toolkit 4. 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (2007)
6. Saroiu, S., Gummadi, P., Gribble, S.: A measurement study of peer-to-peer file sharing systems. Proceedings of Multimedia Computing and Networking (2002)
7. The Raja Project: http://raja.sourceforge.net (2007)
8. Creare Inc.: Ring buffered network bus, http://rbnb.creare.com/RBNB (2007)
9. The NEESGrid Project: http://it.nees.org/ (2007)
10. Apache Nutch Project: http://nutch.apache.org/nutch (2007)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150 (2004)
12. Beckett, J.: Scaling IT for the Planet, http://www.hpl.hp.com/news/2001/oct-dec/planetary.html January (2002)
13. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. In: Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18) (2001)
14. Djilali, S., Herault, T., Lodygensky, O., Morlier, T., Fedak, G., Cappello, F.: RPC-V: toward fault-tolerant RPC for internet connected desktop Grids with volatile nodes. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (2004)