HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4

Li Qi[1], Hai Jin[1], Ian Foster[2,3], Jarek Gawor[2]
[1]Huazhong University of Science and Technology, Wuhan, 430074, China
quick@chinagrid.edu.cn; hjin@hust.edu.cn
[2]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.
{foster, gawor}@mcs.anl.gov
[3]Department of Computer Science, University of Chicago, Chicago, IL 60637, U.S.A.

Abstract—*Grid computing is becoming more and more attractive for coordinating large-scale heterogeneous resource sharing and problem solving. Of particular interest for effective Grid computing is a software provisioning mechanism. We propose a highly available dynamic deployment infrastructure, HAND, based on the Java Web Services Core of Globus Toolkit 4. HAND provides capability, availability, and extensibility for dynamic deployment of Java Web Services in dynamic Grid environments. We identify the factors that can impact dynamic deployment in static and dynamic environments. We also present the design, analysis, implementation, and evaluation two different approaches to dynamic deployment (service level and container level). And examine the performance of alternative data transfer protocol for service implementations. Our results demonstrate that HAND can deliver significantly improved availability and performance relative to other approaches.*

1. INTRODUCTION

A Grid is an Internet-connected computing environment in which computing and data resources are geographically distributed in different administrative domains, often with separate policies for security and resource use. With the introduction of OGSA [11], the focus of Grid computing moved from legacy computing-intensive applications to service-oriented computing based on open standards. Globus Toolkit (GT) development has tracked this trend, with GT4 [10, 12] building on the Web Service Resource Framework (WSRF) specifications to provide an efficient, extensible, stateful, and flexible Grid middleware.

Experience within ChinaGrid [1, 9] and elsewhere [2, 17] has emphasized the importance of dynamic service deployment and management as an enabler of dynamically extensible virtual organizations (VOs) [8]. More specifically, we identify the following requirements when services are hosted on a dynamic management-enabled Grid:

- Services must adapt at run time to changes of scale in VOs and to changes in the number of users.
- It must be easy to reconfigure, redeploy, and undeploy services, without shutting down the whole system.
- Grid software provisioning must take into account dynamic and unpredictable service demand from VO members.
- The availability of target management units for service requests should be maximized.
- Dynamic features should result in minimal development and management costs; that is, users must not be required to implement numerous interfaces or rules for dynamic deployment features.

Dynamic deployment is a big challenge: frequent shutting down and starting up of services for software upgrades or changes in resources or users can increase management costs. Some dynamic deployment solutions have been proposed based on Apache Tomcat's dynamic deployment functionality [2, 3, 14]. However, that infrastructure provides poor performance and availability, with the consequence that (for example) dynamic deployment during a workflow's execution can cause a critical task to fail. In another scenario, if a deployment operation is delayed or canceled because the target container is unavailable, the dependent task would also be delayed or canceled.

Before discussing the design and implementation of our dynamic deployment infrastructure, we introduce some basic concepts:

- *Containers* in a Web Services-based system such as GT4 host *services* and execute *user requests* issued by clients that invoke operations defined by those services.
- The term *dynamic deployment* denotes the ability for remote clients to request the *upload and deployment of new services* into, or the *undeployment of existing services* from, existing containers.
- Issues of *correctness* and *performance* are of particular importance when service requests and deployment requests occur concurrently.

We define *availability* in a dynamic deployment infrastructure as *the proportion of time a system is in a functioning condition*. The meaning of "functioning" is different for two types of clients.

- End users of services deployed in the container care about the success rate of their requests and the response time of those requests. Key issues for these users are the extent to which a container becomes inaccessible during dynamic deployment, and any overhead imposed on ordinary requests by the dynamic service infrastructure (e.g., due to locking).
- Users who deploy services, typically administrators, care about both the success rate and average time cost of the dynamic deployment requests themselves.

Based on these considerations, we adopt as metrics the success rate of user requests, the average time cost for deployment requests, and the deployment availability.

To implement highly available and capable dynamic deployment functionality, we propose HAND (*Highly Available dyNamic Deployment).* The design of HAND is intended to meet six criteria:

1. A container that receives a dynamic deployment requests should complete existing user requests if possible, while ensuring that the dynamic deployment request is accepted as scheduled.
2. User requests received during execution of a dynamic deployment procedure should be handled correctly if possible.
3. When a dynamic deployment procedure is finished, user requests for newly installed services should be handled correctly.
4. The deployment procedure should be decomposed into smaller steps to reduce the risk of deadlock. Generally, deadlocks in the dynamic deployment procedure arise when threads use common runtime resources (e.g., the common ClassLoaders) concurrently. Simplifying and decomposing the deployment steps into smaller substeps can reduce the time that the deployment threads and ordinary threads occupy shared resources.
5. Multiple redundancy approaches should be provided to both remote and local users in order to reduce unavailability. If one approach proves to be unavailable, a user or container can adopt other approaches as backup to finish the deployment.
6. The performance of the deployment procedure should be optimized to decrease the possibility of conflict between ordinary and deployment requests. In short, the overhead caused by the dynamic deployment infrastructure should be as small as possible to the ordinary requests.

In the following, we explore how our performance metrics can be maximized and these criteria met via two different approaches to dynamic deployment:

- *Service-level* deployment (HAND-S), in which we deactivate one or more existing services, install new services, and re-activate those services—without reloading the whole container.
- *Container-level* deployment (HAND-C), in which the installation of any new service involves reloading (reinitializing and reconfiguring) the whole container.

We shall describe implementation techniques for both approaches and present experimental results that demonstrate that when HAND works concurrently in a dynamic network environment, it can deliver capability and availability that are acceptable and that meet the criteria described above.

The rest of this article is as follows. In Section 2 we describe the HAND architecture and implementation. In Section 3 we show how users can utilize the different deployment levels and approaches to improve the availability and capability of dynamic deployment. In Section 4 we present our evaluation of the implementation. In Section 5 we discuss related work. In Section 6 we conclude with a brief discussion of future research.

2. HAND DESIGN AND IMPLEMENTATION

Provisioning is an important feature in cluster computing and has been included in the OGSA specification [11]. However, the GT3 and GT4 releases of the Globus Toolkit Java Web Service Core have not addressed dynamic service deployment.

Weissman et al. [2] have implemented dynamic deployment in GT by building on the dynamic deployment-enabled Apache Tomcat server as the hosting environment. With this approach, users package their services with the basic Java WS Core libraries as a WAR file and deploy it into Tomcat as a Web application. With the help of Tomcat Manager the user can redeploy this application dynamically without restarting Tomcat or interfering with other Web applications.

An alternative approach (adopted here) is to refactor the kernel structure of the Java WS Core standalone container. This approach is more complicated as it requires low-level changes to the container. However, we gain the benefit of a more lightweight dynamic deployment implementation and simpler management. With this approach, we can use the Grid archive (GAR) format for services and reuse GT's existing deployment mechanism. The result, as we shall show, is a highly available dynamic deployment infrastructure.

Weighing these advantages and disadvantages, we designed and implemented HAND in three parts, as illustrated in Figure 1. The shaded components—Service Package Manager, Auto Deployer, and Local Directory—are optional; they are provided to support future advanced provisioning features.

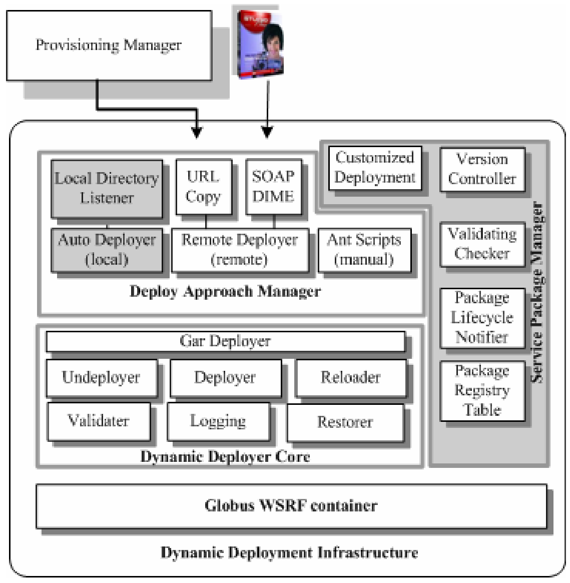

Figure 1 Dynamic deployment modules

2.1 Dynamic Deployer Core

The Dynamic Deployer Core (DDC) is the kernel part to realize the dynamic deployment in HAND and meet the criteria described in Section 1. The challenge in the DDC is to resolve two factors:

1. The deactivation and activation of the services and container.

2. The update of the runtime context in JVM, especially in case of dynamic and multiple ClassLoaders.

To address time cost and safety issues, we use two ClassLoaders to isolate manageable services from system services. The *common ClassLoader* is responsible for the basic libraries used to run the container such as the XML parser, the SOAP engine, logging, and security; this ClassLoader is not reloadable. The *service ClassLoader* is responsible for loading service libraries and is fully reloadable.

As shown in Figure 1, DDC is comprised of seven parts. The *GAR Deployer* is in charge of invoking the actual deployment actions. In the HAND, a reloading action can be safely executed once all services are shut down but before a new service ClassLoader is obtained. The *Undeployer*, *Reloader*, and *Deployer* are reloading actions that can be passed from the *Deploy Approach Manager* to the GAR Deployer. Reloading action can update the service libraries, configuration files, and so forth. The *Validater* is responsible for checking the correctness of the GAR file that is being deployed. It prevents DCC from deploying invalid or malformed GAR files. The *Logging* module is used to record a detailed log of the execution of the reloading actions. Finally, the *Restorer* is a simple backup mechanism. In case of an error it can help the container to restore to its previous working state.

2.2 Deploy Approach Manager

The Deploy Approach Manager (DAM) takes as input a GAR file in Java archive format. This file consists of a Web Service deployment descriptor (WSDD), WSRF resource descriptor files, and application implementations and stubs. The GAR file might contain zero or more services.

DAM addresses our desire for redundant approaches (Criteria 5) by extending GT4 to provide three approaches to service deployment.

The ***Auto Deployer*** is an experimental component that allows users to deploy and undeploy a target GAR file into a container by copying it into a specific directory on the local file system. This feature is convenient for local administrators.

The ***Ant Scripts*** are intended for the offline approach of deploying and undeploying the GARs. This approach works only when the container is off. This restriction is necessary in order to prevent conflicts that could arise when deploying into a running container.

The ***Remote Deployer*** is a standard WSRF service deployed in DDC. Named DeployService, this service supports five operations: upload, download, deploy, undeploy, and reload. The first two operations provide two different approaches to transferring a GAR file that is to be deployed to the container:

1. Via SOAP with attachments using the *upload()* function. The GAR file is attached to the request.
2. Via the *download()* function. The request specifies a URL (HTTP/S, GridFTP/FTP, etc.) for the GAR file, and the DeployService uses globus-url-copy to copy the GAR file from that URL location to the local file system.

Once a GAR file is transferred, the DeployService returns an identifier for the GAR. The GAR file can then be deployed by calling the *deploy()* function with that identifier. Once a GAR file is deployed, the DeployService deletes the file automatically.

A deployed service can be undeployed via the *undeploy()* operation. In addition, a client can reload the entire container by invoking the *reload()* operation, which restarts the container without executing any deployment actions. This operation is useful when a service or container configuration has changed.

The DeployService publishes two resource properties:

1. *Undeployable*: a list of GAR identifiers that can be undeployed.
2. *Deployable*: a list of GAR identifiers that have been transferred to the service but not yet deployed.

GSI authentication and authorization are used to ensure that only authorized clients can invoke the DeployService operations.

2.3 Service Package Manager

The optional Service Package Manager (SPM) provides higher-level management features. In particular, the *Package Lifecycle* and *Package Registry Table* maintain information necessary for the service-level implementation. Since each GAR file may contain several services, service-level management cannot be based on a simple GAR file management system. In our service-level implementation, SPM communicates with DDC to manage the target services dynamically. The SPM also includes three components—Version Control, Customized Deployment, and Validating Checker—that complete the complicated service-level Grid software version control and online upgrades in different VOs.

- *Version Control* is responsible for the version management of different services. It also provides metadata about cross-dependencies for the container system. It avoids upgrades of different applications' JARs.
- *Customized Deployment* permits remote users to submit their own deployment scripts; for example a user can deploy an RPM package to a target system.
- The *Validating Checker* is similar to the DDC Validater. A minor difference is that it focuses on more complicated dependencies and conflict checking among different services before deployment.

3. SERVICE-LEVEL VERSUS CONTAINER-LEVEL

We define two approaches which is also the effort to meet Criteria-5 to dynamic deployment:

- In *container-level deployment* (HAND-C) the entire container is reloaded; that is, all services in the container are deactivated and re-activated.
- In *service-level deployment* (HAND-S) a single service that is being deployed or undeployed is deactivated and re-activated. All other services are unaffected.

We have implemented both approaches within GT4. The container-level implementation is complete and has been merged into the GT code repository[1]. The service-level implementation is a prototype, suitable for performance studies but not yet for production use.

Both approaches are important and useful in different scenarios, as we now discuss. Container-level deployment works well when a global upgrade or configuration is needed, while service-level deployment is more flexible and available in dynamic environments.

3.1 HAND-C: Container-Level Deployment

Container-level deployment proceeds as follows to reload a service implementation:

1. Put the container in "reloading mode." The container will then return "service unavailable" error to any request that the container receives during the deployment. This step blocks until all currently executing requests finish or until a specified timeout expires (whichever occurs first).
2. Stop and deactivate all services, resource homes, and so forth.
3. Perform cleanup operations to flush caches that might contain references to the resources and classes loaded by the original deployment.
4. Execute the deployment or undeployment scripts.
5. Reload the whole container. Configuration descriptor files, etc. are re-read, and all services, resource homes, etc., are re-activated.
6. Return the container to the normal operating mode, and start accepting new requests.

This algorithm—in particular,the use of timeouts in Step 1—seeks to balance the demands of Criteria 1 above with container stability. Steps 2 and 3 are executed to address Criteria 3. We note that this algorithm does not address Criteria 2: all user requests to a container are refused during any dynamic deployment operation on that container.

The timeline in Figure 2 depicts a typical deployment operation. In this figure, "Service Session" denotes an execution of an ordinary request and "Deploy Session" the execution of a deployment request. Moving from the top down, we see first a request that is interrupted due to the Step 1 timeout; then three requests that complete successfully against the old service version; then the deployment request; then three requests that are refused because deployment is in progress; and finally two requests that execute successfully against the new version of the service.

This approach is similar to that used in the Tomcat container. It has the following advantages:

- It avoids deadlock for it does not care about the dependencies among the services deployed in container. It just reloads the whole container.
- It works well when we need to reload the whole service container, including the global configuration, service handlers, and providers. In case of a global installation and configuration issued or service-level reload operation failed, the container-level reload is significant to promise the container keeping stable.
- It minimizes memory and time costs in management, since all services share the same runtime context, i.e. the common service ClassLoader, unified deactivating and activating procedures, and GAR management.

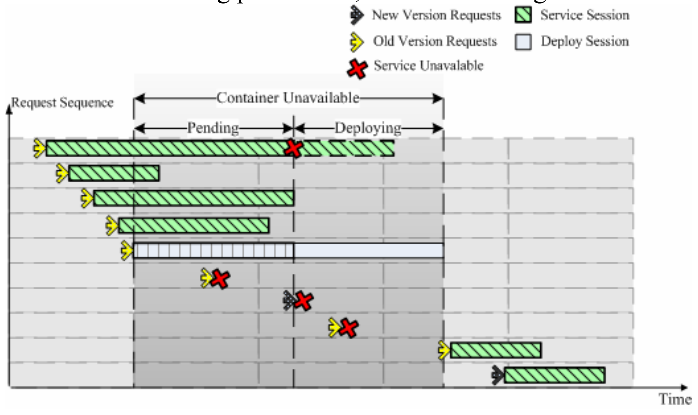

Figure 2: Container-level deployment available sequence

On the other hand, container-level deployment has disadvantages:

- (Re)deployment of *any* service results in the loss of nonpersistent state associated with *all* services. While arguably no service implementation should make any assumptions concerning the availability of nonpersistent state, in practice people often write services with such assumptions in mind. Thus, dynamic deployment can result in unpredictable behaviors for clients.
- Deployment time is unpredictable when several parallel threads are involved (see Section 4.3).

3.2 HAND-S: Service-Level Deployment

Service-level deployment requires complete service isolation (including the service JAR files), a hierarchical ClassLoader tree (a separate ClassLoader for a set of services associated with a GAR file), and an SPM to manage the separate services. This approach allows us to address Criteria 2 of Section 1: the reduction in reloading granularity means that the deployment procedure need not impact requests to other services unrelated to those that are being deployed. Our service-level deployment logic meets these requirements as follows:

1. Check the requested target service name. If it matches the DeployingService, then switch the ClassLoader to system level, or use the normal services' own ClassLoader registered in the SPM. If requests are being processed that involve the services that are to be deployed, then suspend deployment until those requests complete or a timeout occurs.
2. Stop the services that are to be deployed if they are already running. During this period, the container will return "service unavailable" to any request to the services in the GAR file.

[1] The stable implementation can be checked out from *:pserver:anonymous@cvs.globus.org:/home/globdev/CVS/globus-packages* with module name 'wsrf'.

3. Stop any services on which the pre-deployed services depend, and deactivate any related resources. Typically, these services are named in the pre-deployed GAR file.
4. Perform cleanup operations to flush caches that might contain references to the classes loaded by the original deployment (just for redeploy).
5. Execute the deployment or undeployment scripts.
6. Create or update the working space context for the new services, and then register to the SPM registry.
7. Initialize, activate, and start the new services.

The main difference between this approach and the container-level approach is that the reloading unit here is the service rather than the container as in HAND-C. Figure 3 shows what happens when requests arrive during dynamic deployment in HAND-S. Requests 1, 2, 3, 4, 6, and 8 are to services other than the services being deployed, and can thus proceed without interruption. Only the 7th and 10th requests are to a service that is being deployed, and thus these two requests fail and succeed, respectively, as they occur during and after deployment.

Service-level deployment has the following advantages relative to container-level deployment:

- The time required to reload a service is more predictable as it depends primarily on characteristics of that service, not other components deployed in the same container.
- Because there is no need to wait for and deactivate all services in the container, service-level deployment is much faster in most cases.

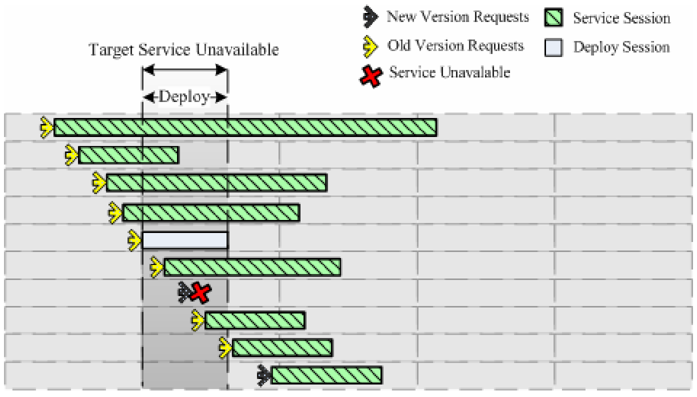

Figure 3: Service-level deployment available sequence

Service-level deployment also has limitations:

- There is a need for additional internal synchronization in the container, which can be costly. The container should switch among different ClassLoaders to match the various service requests, and must also maintain consistency with persistent storages (a XML file in our implementation), JNDI resources, and Caches existing in service instances. Furthermore, HAND-S cannot handle circular dependencies among services well. (The dependency of service composition is another critical challenge in Grid which will be discussed in our future papers.)
- The need for a more detailed registry results in increased memory usage and execution time costs.
- If the registry structure is destroyed or the global configuration is updated, we must use container-level deployment to reinitialize the whole container.

3.3 *Time Cost Analysis*

We now discuss the costs associated with the two deployment approaches. We use the symbols in Table 1.

TABLE 1 SYMBOLS USED IN THIS PAPER

Notation	Definition
t_{total}	Total time required to deploy a target GAR file.
$t_{transfer}$	Time required to transfer the GAR file.
$t_{pending}$	Time required to wait for the container to become available.
t_{deploy}	Deployment time for script processing during deployment.
t_{reload}	Time required to restart container or services.
t_{limit}	Reload timeout limitation for a running service.
t_{system}	System cost to reload the container itself.
t_i	Time to stop and deactivate target service i.
t'_i	Time to activate and start target service i.
s_i	Execution time left for unfinished request i.
R	*Running* services that are requested during dynamic deployment and are being processed on the container.
D	*Deployed* services on the target container; these are the aggregate of all the services deployed on the target container, whether activated or deactivated.
D′	Services that are prepared to be deployed.

To meet Criteria 4, as shown in Figure 4, the deployment procedure in HAND consists of several phases: *deployment preparation*, *physical deployment*, and *system reloading*. The dashed "Pending" box indicates that it depends on the concrete approach (HAND-S or HAND-C) chosen to issue the reloading.

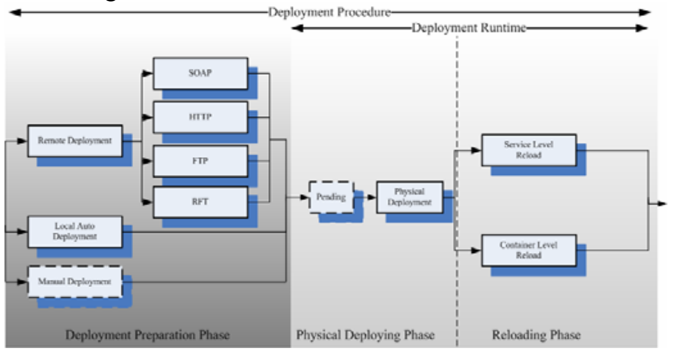

Figure 4: Three-phase deployment procudure

The total dynamic deployment time consists of four parts:

$$t_{total} = t_{transfer} + t_{pending} + t_{deploy} + t_{reload} \qquad (1)$$

The t_{deploy} and $t_{transfer}$ components are determined by the deployment methods used in the preparation phase and the complexity of the ANT scripts that implement deployment actions, both of which are independent of the deployment procedure.

For HAND-C, $t_{pending}$ and t_{reload} are as follows.

$$t_{pending} = \min(\max_{R}(s_i), t_{limit} \times \|R\|) \quad (2)$$

$$t_{reload} = \sum_{i=1}^{n} (t_i + t_i') + t_{system}, n = \|D\| \quad (3)$$

Based on the discussion in Section 3.2, we conclude that the pending time is mainly spent waiting for completion of existing requests. Hence, the total time is the minimum of the maximum remaining time of the currently executing requests and the time required to interrupt for all running threads. The reloading time is equal to the system reloading time plus the deactivation and activation time for all deployed services.

For HAND-S, the pending time is the time spent waiting for completion of existing requests for the target service and any related services. (However, we note that the problem of service dependency is complicated; we will investigate it and discuss it in future papers.) Here, we assume that related services D' are just the services defined in the GAR's Web Service Deployment Descriptor (WSDD) file. The reloading time shrinks to the sum of the deactivating time and activating time of the related services.

$$t_{pending} = \mathbf{min}(\mathbf{\max_{D' \cap R}}(s_i), t_{limit} \times \|D' \cap R\|\mathbf{)} \quad (4)$$

$$t_{reload} = \sum_{D' \cap R} t_i + \sum_{D'} t_i' \quad (5)$$

We define t_{total}(HAND-S) and t_{total}(HAND-C) as the time cost for two approaches, and we assume that both approaches use the same deploying method and the same GAR file. The relationship between running services, predeployed services, and deployed services is

$$D' \cap R \subseteq R \subseteq D \quad (6)$$

If the processing requests finished at the same time, then $t_{pending}$(service) $\le$ $t_{pending}$(container). Similarly, we can achieve t_{reload}(service) $\le$ t_{reload}(container).. Hence, it is not too difficult to conclude that t_{total}(service) $\le$ t_{total}(container) in a dynamic invocation environment.

4. EVALUATION

A comprehensive evaluation of dynamic deployment is challenging because of the difficulty of capturing the complexities of a realistic Grid environment. Thus, we focus on micro-benchmarks designed to capture specific aspects of dynamic deployment behavior, namely, deployment time, capability and availability, and file transfer performance.

4.1 Dynamic Deployment Experiments

As discussed in Section 3, the service container becomes unavailable during dynamic deployment. Thus, we first measured deployment time as a function of both the size of the file being deployed and the number of services in the container.

The experimental setup was as follows. We installed and tested the HAND containers at two sites: a local site with three PC servers, powered by Pentium 4 2.4 GHz with 2 GB RAM and 37 GB hard disk in a cluster; and a remote site equipped with one PC server, powered by Pentium III 1 GHz with 1024 MB RAM. The two sites were connected by a 2.5 G fabric WAN shared with other CERNET applications. The local cluster was connected with 100 Mb Ethernet. All tests ran on Fedora 3 with Linux kernel 2.6.9-1.667. The Java version was J2SDK 1.4.2_08-b03, and we used –Xms64m and –Xmx1024m parameters to enlarge the maximum JVM memory.

We constructed a set of Grid Archive (GAR) files for use in our experiments, as summarized in Table 2. The first five files ranged in sizes from 42 KB to 100 MB, a range typically encountered in Grid applications. We also constructed a large number of identical services, testService_clone0 to testService_cloneN, for evaluating the impact of the number of deployed services on performance. The *Jar Scale* column in Table 2 denotes how many JAR files were in the package; the *Compression Rate* is the ratio of compressed file size to the original.

TABLE 2 TEST PACAKGES USED IN OUR EXPERIMENTS

Id	Package Name	File Size (KB)	Jar Scale	Compression Rate (%)
1	testService1.gar	42	2	63
2	testService2.gar	1,154	4	33
3	testService3.gar	12,909	56	90
4	testService4.gar	42,236	61	75
5	testService5.gar	98,004	59	55
6..n	testService_clone N.gar	40	2	63

We measured the time required to deploy each of files 1-5 of Table 2 into a container running nine services—the basic service set deployed by default by GT. The only request made during the dynamic deployment procedure was that to *DeployService*. Each deployment was repeated 40 times. We present the deployment and reloading times (as discussed in Section 3.3); the pending time is zero in this case.

Figure 5 gives our results. Timers in our implementation allow us to break down the total deployment time into the following categories:

- Deploy (D), which includes the physical deployment time (D-D) and reloading time (D-R) as discussed in Section 3.3.
- Redeploy (R), which includes the physical redeployment time (R-D) and reloading time (R-R).
- Undeploy (U), which also includes the physical undeployment time (U-D) and reloading time (U-R).

The time required to execute physical deployment scripts is the biggest cost in the Deploy operation (D); the physical redeployment time (R-D) cost is slightly greater than that. Each redeploy operation consists of a sequence of undeploy and deploy operations. In HAND-C, the first step of deployment is to check whether or not the new GAR is already deployed. Undeploy simply deletes all deployed files and reloads the container; the time cost is decided mainly by U-D, which naturally increases with GAR file size. Thus, we can determine that the reloading time, equal to the difference between the operation times (D/R/U) and the physical deployment times (D-D/R-D/U-D), increases slowly as the

GAR size changes. Initially, the biggest time cost of an atomic operation is just less than 20 seconds, which we believe is tolerable for most Grid applications.

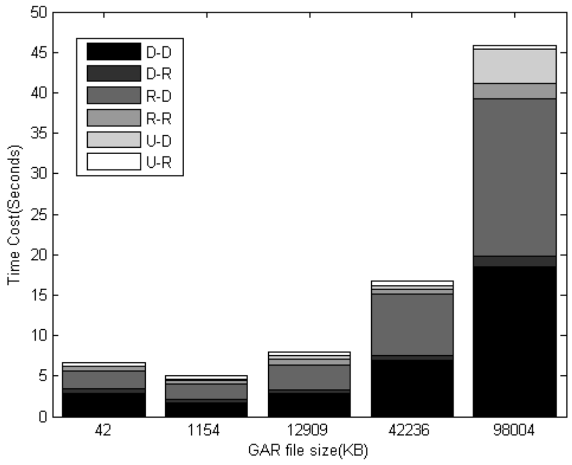

Figure 5: Operation comparisons

To identity additional impact factors on deployment and reloading costs, we enlarged the deployed service scale from nine to 879 services. Figure 6 shows the results of this new experiment. In this figure, the z axis expresses the time cost for reloading the container. The results show that reloading time increases with the number of services. In the service scale, the reloading time of different-sized GAR files is nearly the same when the GAR file size is less than 42 MB. However, the time increases rapidly when the GAR file size increases to about 100 MB. The results show nearly the same linearity in smaller GAR files (42 KB to 42 MB), but become bumpy when the GAR file size increases to about 100 MB. The reason is that the JVM's garbage collector runs in the background randomly. When the deployed service size is big enough, garbage collection shares the reloading time with the HAND core. Generally, the trend should be increasing. Even in the vertex, the reloading time is beneath 30 seconds, which will satisfy the requirements of most Grid applications in a static environment. Accordingly, our service-level implementation is affected less by garbage collection, even for the biggest GAR file. In addition, it costs less time to finish the reload operation.

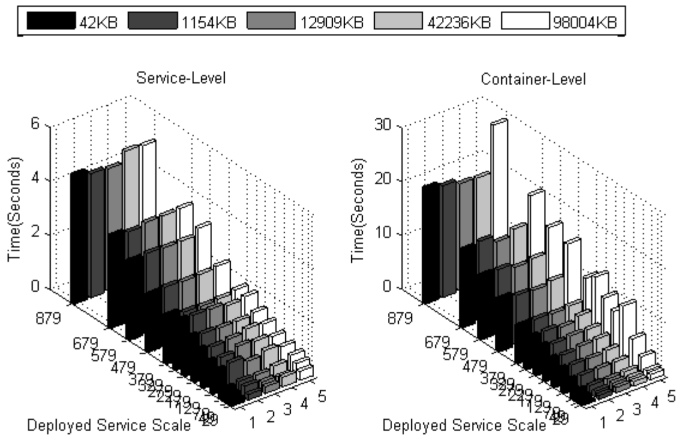

Figure 6: Scaled comparison on reload

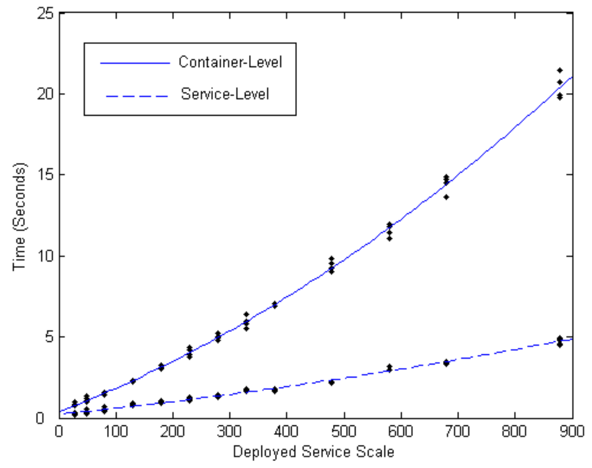

Figure 7: Least Quadratic fitting curve for reload operation

Based on the results from Figure 6, we discard the bumpy container-level data and then attempt to fit a curve to our two sets of data. As described in [21], we define *x* as the deployed service scale and y as the reload time cost, and assume *k*th degree polynomial as:

$$y = a_0 + a_1 x + \dots + a_k x^k \qquad (7)$$

The residual is given by

$$R^2 = \sum_{i=0}^{n} [y_i - (a_0 + a_1 x_i + \dots + a_k x_i^k)]^2 \qquad (8)$$

By using least quadratic fitting technique, we filled in our data, and finally achieved when k equals 2, we could get the fittest polynomial for two levels. As formulas list below, y_c is the container-level time cost polynomial and y_s is the service-level.

$$y_c = 0.0106x^2 + 13.478x + 372.5, R^2 = 0.997 \quad (9)$$

$$y_s = 0.0018x^2 + 3.5468x + 235.86, R^2 = 0.991 \quad (10)$$

Figure 7 denotes our fit curves. We see that service-level reloads are less expensive than container-level reloads in all circumstances. This result confirms our earlier analyses and matches our conclusions in previous sections. We note that the availability of sufficient memory for service-level deployment is an important precondition for this result.

From formula 9 and 10, we see that if no services are deployed, both approaches incur a reload cost of around 300ms. And in both cases, the polynomial time are all $O(x^2)$.

To enable direct comparison with other dynamic deployment enabled containers, we repeated our experiments on the Apache Tomcat container (version 5.0.30), as used by Weissman et al. [2, 3] and Matthew et al. [17]. (However, we note that we use GT4, not GT3 as used by those authors.) Also, we increased the JVM memory to 1G by adding the '-Xms64m –Xmx1024m' parameter. Figure 6 presents the experimental results in the different service-level (10, 200, 400, and 600 services) environment. In this figure, HAND-C is our container-level implementation, HAND-S is the service-level implementation, and TOMCAT denotes the Apache Tomcat hosting environment.

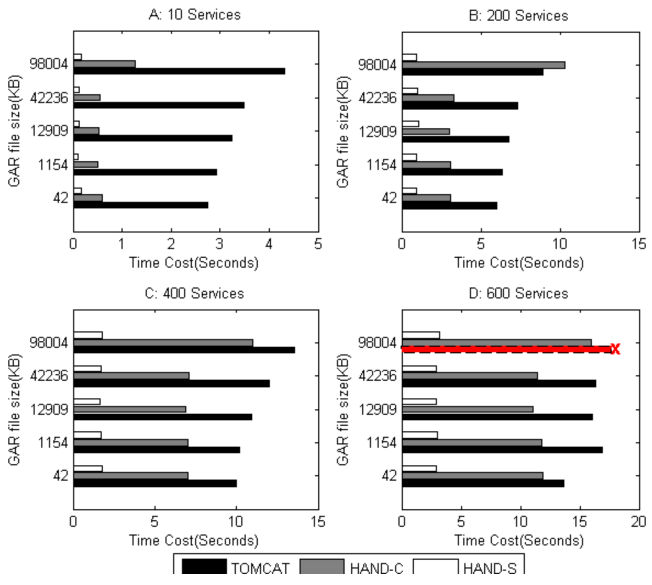

Figure 8: Comparisons against the Tomcat container

We see in Figure 8 that both HAND-C and HAND-S use less time to reload than does Tomcat, particularly when few services are deployed. (The one exception is in 8(B), when HAND-C is slightly slower than Tomcat for the largest file.) This effect is particularly evident in the case of our service-level implementation (Tomcat is a container-level implementation), but is also evident in the case of the container-level implementation, presumably because Tomcat typically deactivates or activates more components, including Apache Axis itself, cluster components, Jasper, and the like. The HAND core, however, mainly involves only Apache Axis and just reloads the ClassLoader and updates the JNDI tree appropriately. In 8-D, Tomcat issues an 'out of memory' exception for the largest GAR file, which is particularly concerning our desire for container stability.

As shown above, if support for massive dynamic deployment-enabled applications is needed, one should use the HAND container instead of Tomcat to guarantee capability, usability, and reliability.

4.2 *Capability and Availability in Dynamic Environment*

Our next experiments were designed to study interactions between service deployment operations and other requests to a container. We designed these experiments as follows:

1) We first dynamically deployed eight cloned services that take, respectively, 0, 30, 60, 90, 120, 180, 240, and 330 seconds to process a user request.
2) We then started four client threads, each of which issued a series of 1,000 invocations, each to one of the eight services started in #1. These threads also logged both failed and successful invocations.
3) We also started a thread that issued a series of 100 deploy requests at random intervals during the period of Step 2. Each such request deployed, redeployed, and undeployed one of a second set of 10 cloned services.

We ran experiments on both HAND-C and HAND-S, and used four parallel threads to do our experiments, which is the high water mark of the GT4 Java core container (i.e., the medium overhead of GT container). We configured the GT4 container with 35 deployed services, which means (based on the results of Section 4.1) that HAND-C should incur a reloading cost of 875ms and HAND-S a cost of 362ms. Finally, we note that al deployed services are unrelated in logic: i.e., services in different GARs do not invoke the others' ClassLoaders.

Figure 9 shows that the deployment time increases rapidly with service serving time for HAND-C, due to the need to wait for the completion of service invocations that are already in progress. When service serving time is long enough (the cross mark in the figure), namely, greater than the reloading timed-out limitation (default 5 minutes in HAND-C), the deployment operation is canceled, due to the client timeout of 10 minutes. In contrast, while HAND-S deployment time is initially slightly higher than for HAND-C, it then stays fairly constant as service serving time increases.

Figure 10 shows that HAND-S also achieves consistently high success rates. This result is as expected, given that the reload of one service does not effect other services in the same container. In contrast, the HAND-C success rate is low and unpredictable. When the service serving time grows above the reload timed-out limitation, most client invocations are canceled because of client timeout.

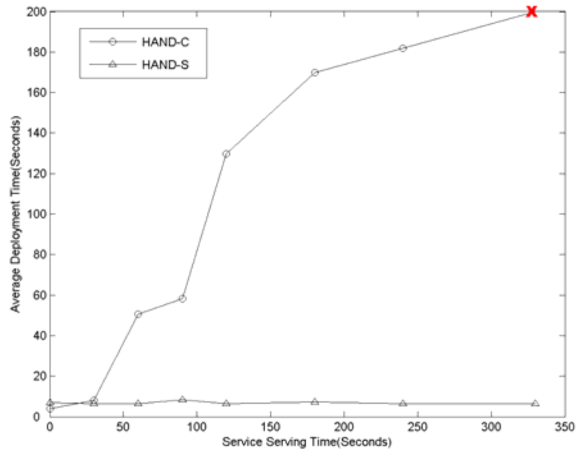

Figure 9: Deployment time in a dynamic environment

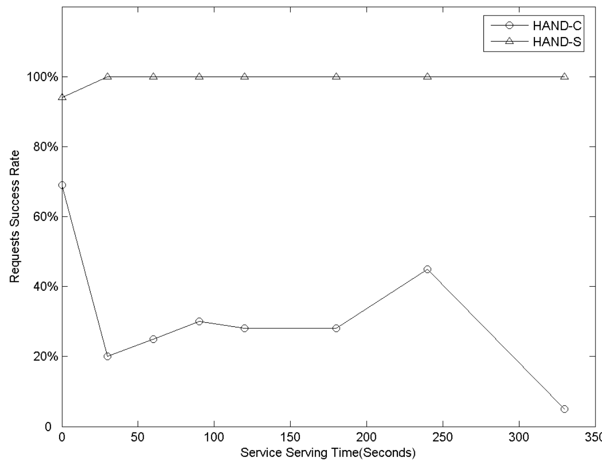

Figure 10: Success rate in a dynamic environment

In Table 3, we show the average deployment time for both HAND-S and HAND-C across these experiments. Our results indicate that the capability of dynamic deployment at the container level is unpredictable in a complicated dynamic Grid environment. The cost stems from promising the success rate of the requests before dynamic deployment and preventing deadlock during container restart. Moreover, the success rate of normal service requests also decreases when deployment is triggered. This level is suitable for clients that incorporate some fault tolerance logic.

TABLE 3 COMPARISON OF EXPERIMENTAL RESULTS

	HAND-S	HAND-C
Average Deployment Time, ms	6849.57	85487.94
Average Success Rate, %	99.25	31.13

We conclude that service-level deployment is more capable and more available than container-level deployment for dynamic and complicated Grid environments.

4.3 GAR File Transfer Performance

Our third set of experiments focused on the performance of the DAM file transfer function. We evaluate the performance of three transfer protocols in LAN and WAN environments. The test packages used the five GAR files, which were invoked 20 times each.

Figures 11 and 12 show that in both the LAN and WAN, the SOAP attachment in DIME format approach costs more than HTTP or FTP. When the GAR file size is near to or less than 10 MB, the SOAP attachment is convenient for users to transfer the target files directly from the client. For larger files, it is advisable to choose GridFTP, FTP, or HTTP. Especially in a WAN environment, transferring big GAR files by SOAP attachment will cost more time and more server memory, and may even cause the server to run out of memory. Compared with HTTP, FTP may provide better access control and flow control. For secure and reliable transfers, the GT4 GridFTP [15] and Reliable File Transfer (RFT) service [16] are recommended, although we note that's GridFTP use of public key authentication imposes a startup overhead relative to traditional FTP.

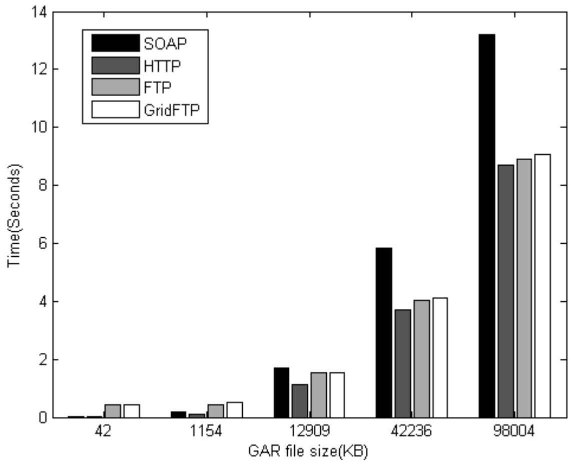

Figure 11: Transfer time comparison, in LAN

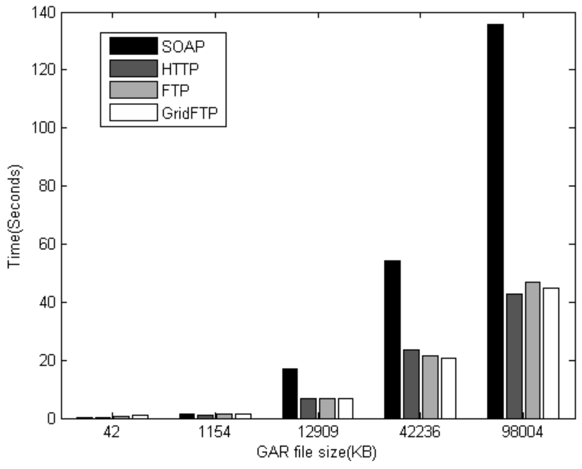

Figure 12: Transfer time comparison, in WAN

5. RELATED WORK

Dynamic service deployment functionality has been explored and developed in many different contexts, including J2EE [18, 19] and Web Services [20].

Rauch et al. [6] implemented partition cloning and partition repositories as well as a set of OS-independent tools for software maintenance using entire partitions, thus providing a clean abstraction of operating system configuration states. However, this approach is not suitable for service-oriented architectures. Moreover, the deployment of an entire OS image is expensive, and the deployment itself will seriously impact system availability. Chase et al. explore related ideas in their Cluster on Demand project [22].

Keahey et al. [4, 5] use virtual machine technology (e.g., Xen, VMware) to build virtual working environments and to provide for the dynamic management of the Grid job life cycle. Their use of virtual machines rather than JVMs to host user computations leads to somewhat different solutions from our service-oriented approach.

ROST [7], deployed in the CROWN Grid, focuses on dynamic and remote deployment for WSRF core with secure access. The developers evaluated remote deployment in the load balancing of local clusters. However, they did not discuss in detail the capability and availability of deployment.

Weissman et al. present an architecture and implementation for a dynamic Grid service architecture based on Tomcat that extends GT3 to support dynamic service hosting (hosting and rehosting a service within the Grid in response to service demand and resource fluctuation) [2, 3]. Their implementation allows new services to be added or replaced without taking down a site for reconfiguration and allows a VO to respond effectively to dynamic resource availability and demand. But the implementation is based completely on Tomcat's container-level deployment capability, which suffers from poor performance.

These and a few other projects [14, 17] are the main dynamic deployment efforts for Grid applications. Some of them clearly are not intended for a WSRF-enabled service-oriented architecture. Moreover, although some have implemented service-oriented dynamic deployment, they do not address in detail the cost, namely, the capability brought from dynamic deployment itself and the availability in dynamic Grid environments.

6. CONCLUSION AND FUTURE WORK

We have described HAND, a highly available dynamic deployment infrastructure for use in the Globus Toolkit Java Web Services container. HAND addresses dynamic service deployment at both the container level and the service level, and thus supports different granularities with different session lock characteristics, applicable for different Grid applications and scenarios. HAND can be adapted to dynamic conditions and changing user requirements. Three factors that affect HAND performance are the size of the predeployed GAR files, the number of services deployed in the container, and the runtime invocations and service serving time during deployment. Experiments show that HAND provides good capability, extendibility, and availability.

We plan to complete a robust implementation of our prototype service-level deployment. We would like to design a mechanism to handle the dependency conflicts among deployed services. Using HAND to enhance Grid software provisioning is a major challenge in the Grid community. We will focus on integrating HAND with the GT information system and workflow, in order to build a real self-configuring, self-curing, and self-propagating Grid system.

ACKNOWLEDGEMENTS

The work of IF and JG was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract W-31-109-Eng-38, and by the NSF NMI program.

REFERENCES

[1] H. Jin. "ChinaGrid: Making Grid Computing a Reality," *Proceedings of ICADL'04*, Lecture Notes in Computer Science, (2004), 3334, 13-24.

[2] J. Weissman, S. Kim, and D. England. "Supporting the Dynamic Grid Service Lifecycle," *CCGrid04*, 2004.

[3] J. Weissman, S. Kim, and D. England. "A Framework for Dynamic Service Adaptation in the Grid: Next Generation Software Program Progress Report," *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[4] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron. "Virtual Workspaces in the Grid," *Europar 2005*, Lisbon, Portugal, September, 2005.

[5] K. Keahey, I. Foster, T. Freeman, and X. Zhang, "Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid," *Scientific Programming*. 2006.

[6] F. Rauch, C. Kurmann, and T. M. Stricker, "Partition Repositories for Partition Cloning – OS Independent Software Maintenance in Large Clusters of PCs," *IEEE International Conference on Cluster Computing*, 2000, 233-242.

[7] H. Sun, Y. Zhu, C. Hu et al. "Early Experience of Remote and Hot Service Deployment with Trustworthiness in CROWN Grid," *APPT 2005*: 301-312

[8] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, 15(3), 2001.

[9] Y. Wu, S. Wu, H. Yu et al. "CGSP: An Extensible and Reconfigurable Grid Framework," *APPT 2005*, pp.292 – 300

[10] M. Humphrey, G. Wasson, J. Gawor, et al., "State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations," *14th International Symposium on High Performance Distributed Computing* (HPDC14), 2005.

[11] Global Grid Forum. "Open Grid Service Architecture, Version 1.0." http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf

[12] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," *IFIP International Conference on Network and Parallel Computing*, 2005, Springer-Verlag LNCS 3779, 2-13.

[13] Apache Axis Group. Developer Guides. http://ws.apache.org/axis/java/developers-guide.html

[14] P. Watson and C. Fowler, "An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet," *Technical Report Series, CS-TR-890*, University of Newcastle upon Tyne

[15] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus Striped GridFTP Framework and Server," *SC'2005*, 2005.

[16] W. Allcock, I. Foster, and R. Madduri, "Reliable Data Transport: A Critical Service for the Grid," *Building Service-Based Grids Workshop*, 2004, Global Grid Forum 11.

[17] M. Smith, T. Friese, and B. Freisleben. "Towards a Service-Oriented Ad Hoc Grid," *3rd International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, 2004, pp.201-208.

[18] F. Reverbel, B. Burke, and M. Fleury. "Dynamic Deployment of IIOP-Enabled Components in the JBoss Server," *Component Deployment: Second International Working Conference, CD 2004*, Edinburgh, UK, May 20-21, 2004. pp. 65 - 80.

[19] N. Sridhar, J. O. Hallstrom, and P. A. Sivilotti. "Container-based component deployment: A Case Study," Technical Report OSU-CISRC-2/04-TR08, Computer Science and Engineering, The Ohio State University, Columbus, OH, February 2004.

[20] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H.H. Ngu. "Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services,". In *18th Int. Conference on Data Engineering (ICDE)*, pages 297–308, San Jose, CA, February 2002. IEEE Computer Society.

[21] Eric W. Weisstein. "Least Squares Fitting--Polynomial." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html

[22] Chase, J., Grit, L., Irwin, D., Moore, J. and Sprenkle, S. "Dynamic Virtual Clusters in a Grid Site Manager". In *12th International Symposium on High Performance Distributed Computing (HPDC-12).* 2003.