Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-041

Supporting the Dynamic Grid Service Lifecycle

Jon Weissman, Seonho Kim, and Darin England

November 17, 2004

Technical Report

Supporting the Dynamic Grid Service Lifecycle

Jon B. Weissman, Seonho Kim, and Darin England

Nov, 2004

Department of Computer Science & Engineering
University of Minnesota
4-192 EE/CS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

Supporting the Dynamic Grid Service Lifecycle

Jon B. Weissman, Seonho Kim, and Darin England
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
(*jon@cs.umn.edu*)

Abstract

This paper presents an architecture and implementation for a dynamic OGSA-based Grid service architecture that extends GT3 to support dynamic service hosting - where to host and re-host a service within the Grid in response to service demand and resource fluctuation. Our model goes beyond current OGSI implementations in which the service is presumed to be "pre-installed" at all sites (and only service instantiation is dynamic). In dynamic virtual organizations (VOs), we believe dynamic service hosting provides an important flexibility. Our model also defines several new adaptive Grid service classes that support adaptation at multiple levels. Dynamic service deployment 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. The preliminary results suggest that the cost of dynamic installation, deployment, and invocation, is tolerable.[1]

1.0 Introduction

Computational Grids are undergoing an evolution. The first wave of Grid computing successfully demonstrated the feasibility of Grids for addressing niche problems in high-end scientific computing based on the emergence of Grid middleware, most notably Globus [5], Legion [9], and Condor [14]. These projects have established a core enabling technology based on low-level resource-centric abstractions, machines, data stores, jobs, etc. The next generation of Grids is focusing on how to "elevate" the level of abstraction to better enable Grid application designers and end-users to solve "real problems". It has been persuasively argued that next-generation Grid applications will be increasingly multidisciplenary, collaborative, distributed, and most importantly, dynamic. The latter implies that static infrastructures will not be adequate since such applications may be assembled on-the-fly to exist only for a transient period of time. Such application environments have been coined *Virtual Organizations* (VO) in which "secure, flexible, coordinated resource sharing among dynamic collections of individuals and institutions is required [6]." Grid services have been proposed as a way to address these issues [7].

When services are hosted on the Grid they must adapt due to the dynamics of the Grid and of the VO users. For example, Grid services must adapt to the dynamic and unpredictable resource availability inherent in the Grid resources upon which they are hosted. Grid services must also adapt to the dynamic and unpredictable service demand from clients within the VO. In prior work, we have developed a class of resource multiplexing algorithms for handling multiple concurrent service requests that addresses unpredictable resource demand for a single static instance of a service on a dedicated resource pool [13][21]. In this paper, we present an architecture and prototype implementation for dynamic Grid services that extends OGSA to better support dynamic VOs. In particular, we address the problem of dynamic service hosting -

[1] This work has been sponsored by the National Science Foundation NGS-0305641 and ITR-0325949, and the Department of Energy's Office of Science under grant DE-FG02-03ER25554.

where to host and re-host a service within the Grid. We also propose several new adaptive Grid service classes that are designed to better capture the dynamics of the Grid. Dynamic service deployment allows services to be added or upgraded without "taking down" a site for re-configuration and allows the VO to respond effectively to changing resource availability and demand including "flash crowds". Our prototype system is based on Globus GT3 and the results indicate that dynamic installation and deployment of services is manageable as well as the overhead of service invocation. The remainder of this paper is as follows: Section 2.0 presents the dynamic service architecture. Section 3.0 describes the adaptive Grid service classes in greater detail. Section 4.0 presents benchmark performance results. Section 5.0 provides related work and Section 6.0 is a conclusion and summary.

2.0 Dynamic Service Architecture

An OGSA-based Grid software stack has the potential to provide a coherent and stable platform for Grid application and tool developers in which the Grid is seen as a collection of application- and system-level Grid services (Figure 1). We take the "top half" of the Grid fabric to be OGSA which provides a basic service framework with common services like factories, repositories, registries, etc. The "bottom half" is OGSI and reflects a specific implementation such as Globus GT3 [8], OGSI.net [19], etc. Grid system services provide core functionality that is required by application-level Grid services. One class of Grid system services we are investigating are those which encapsulate and provide resources to enable application-specific Grid services to run. For example, an application-level parallel solver service would need to be "hosted on" a Grid system service that provided CPU resources. To enable VOs to evolve, scale, and respond to unknown events and unpredictable service demands, we believe that dynamic service deployment is needed both for Grid system services and Grid application services. Furthermore, we believe that each service class must be adaptive - an issue that we address in the next section. Our dynamic service architecture consists of several core services and components (Figure 2). The adaptive Grid service (AGS) is our fundamental abstraction for a Grid service that can adapt to changes in demand and resource availability. The AGS consists of three components: a front-end, deployer, and back-end. The AGS front-end handles client requests and makes decisions about where the request should run. The AGS deployer decides which site(s) should host and deploy the service. Information about the service when it is deployed and running is maintained by the front-end. The back-end consists of an AGS factory that contains the actual code for the service and serves each request by creating an instance we call the AGSI (adaptive Grid service instance). OGSA supports both transient and persistent instances and the back-end can be config-

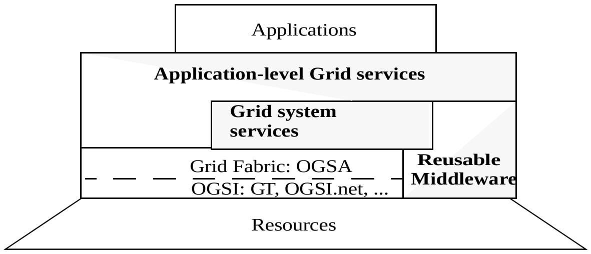


Figure 1: Grid Stack. The bold boxes are addressed in our system.

ured by the service provider to create either type of instance. The back-end is dynamically deployed or hosted using a negotiated and leased pool of resources provided by an adaptive resource provider service (ARP). The leasing model conforms to the OGSI lifetime specification. For negotiation we ultimately plan to support the OGSI agreement specification [4].

There may be "replicas" of the service back-end hosted on different ARPs in the Grid. The service provider creates a front-end and back-end using code templates and runs a packaging tool to create a service package. An installer service can then be run to install the AGS front-end and AGS deployer from this package. In principle, the front-end could be installed on any site in the Grid that is running an ARP. Multiple front-ends can be installed to avoid access bottlenecks. Deploying the service is a 2nd step and is performed by invoking the AGS deployer to initiate deployment of the back-end also from the service package. This requires the selection of a remote ARP on which to deploy service. Once a service is deployed, all front-ends are automatically registered with a registry for future client lookup.

The container provided with GT3 is limited to handling statically deployed services. Once the container is started, no other services can be added. To enable true dynamic deployment, we have used the Tomcat Web container as a replacement (Figure 2). Tomcat provides an API that allows new web applications to be installed *while* the container is running. In the current GT3 platform, newly developed services cannot be deployed and activated without restarting the service container. In order to deploy a Grid service on a remote machine dynamically, the Grid service code is packaged as a web application in a web application archive (WAR). The WAR file is a Java archive file (JAR) and contains web application codes, the web application deployment descriptor, and other related libraries in a directory-structured layout. The Tomcat manager interface allows the packaged web application to be installed and activated dynamically from the WAR file. When packaging a Grid service in WAR file, GT3 related libraries, schema files (WSDL files), and the service deployment description (WSDD) need to be included, in addition to the service codes. The use of Tomcat is only required for sites that wish to support dynamic service deployment, e.g. sites that run Grid system services (ARPs) or local installer sites.

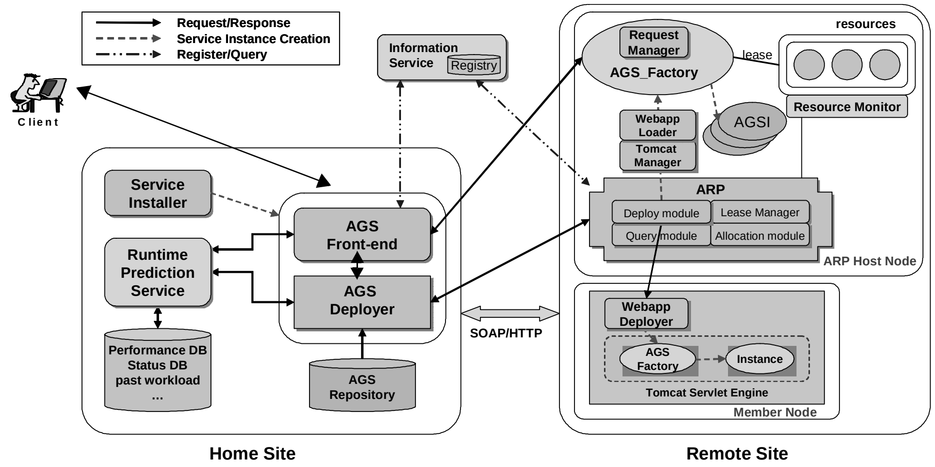


Figure 2: Dynamic Service Architecture

The specific APIs for the architectural components are shown in Figure 3. The *LotType* defines a resource lot in units meaningful to the type of resources provided, e.g. some number of CPUs or some amount of storage. The *LotType* also contains information about the resource lease (*LeaseType*) and enough information about the contained resources (*ResourceType*) to enable the service implementation to deploy/use them. The *ServiceType* defines the string representation of a service, a unique URI-addressed ID. It contains a GSH (Grid service handle) and a GSR (Grid service reference in WSDL format). Lastly, the *PackageType* contains the actual service code and any associated data files required for deployment including WAR files. It is serialized to enable SOAP transmission. The dynamic deployment lifecycle is shown below (Figure 4). We assume that some ARPs are deployed in the VO as part of its core infrastructure (since without resource providers no other services can be deployed!). A client looks up the AGS front-end in a registry and issues a service request to it which will be routed to a back-end. This has the advantage that detailed performance information can be collected as the request is being processed (e.g. it enables the front-end to decide where to send the request based on its parameters), and also provides a simple “one-step” interface to the application.

```
installer {

    ServiceType install (ARP, PackageType); // install AGS front-end service package on ARP
    void uninstall (ServiceType); // uninstall service front-end }

AGS_Deployer {

    ServiceType deploy_AGS (PackageType); // deploy back-end service on a selected ARP
    void undeploy_AGS (ServiceType); // undeploy service back-end
}

AGS_front_end {

    void add_new_AGS (ServiceType, LotType); // inform front-end about new back-end AGS
    void remove_AGS (ServiceType); // inform front-end that a back-end AGS has been removed
    PerfDataType get_perf_data (); // returns performance data for the AGS
    // service-specific interface ... }
```

Figure 3: Core component APIs

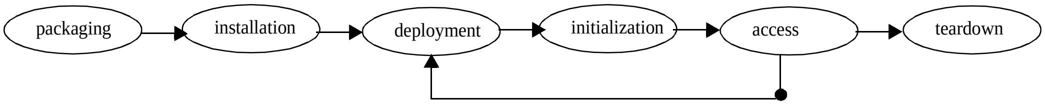


Figure 4: Service Lifecycle. All elements of the service lifecycle are supported in the architecture. Note that dynamic service access may induce additional back-end deployments.

3.0 Adaptive Service Classes

An adaptive service is a Grid service but with special features and interfaces to account for and expose the internal adaptation performed within the service. Adaptive services are divided into system and application services (see Figure 1). A system service performs a generic function such as to manage a pool of resources, e.g. an adaptive resource provider for CPUs or storage (ARP). System services are used by higher-level application services. An ARP implements a policy that defines to what extent it is willing to provide resources to the Grid, how to prioritize between Grid and non-Grid users, and between different

Grid services. For “structured” ARPs with a defined “owner” or provider, such a policy could be based on some form of payment or compensation. Other Grid researchers have explored economic models for Grid resource allocation [22], and on Grid resource accounting [15] that could be used to construct such policies. In this paper, we assume that resources are “leased” for a period of time, and the cost is proportional to the lease length.

Some ARPs may be highly specialized – e.g. an adaptive storage provider (ASP) may only provide long-term storage, an adaptive cpu provider (ACP), may only provide computational resources. An ACP must provide sufficient resources to enable execution including short-term use of memory and scratch storage. In this remainder of this paper, we focus on ACPs and use the term ARP and ACP interchangeably. An adaptive Grid service (AGS) denotes a specific application-level service. We focus on application-level services that are on the high-end in terms of resource requirements, e.g. a parallel equation solver, since these would be more typical in a Grid environment. High-end services represent an important class of Grid services that can be broadly defined as requiring significant resources in terms of computation, communication, or data storage. We also believe that these services are most sensitive to the dynamism inherent in the Grid.

3.1 AGS

High-end Grid services are particularly attractive for a variety of reasons. They allow the user to focus on their application and obtain remote service when needed by simply invoking the service across the network. The user can be assured that the most recent version of the code or service is always provided and they do not need to install, maintain, and manage significant infrastructure to access the service. For high-end applications in particular, the user is still often required to install a code base (e.g. MPI), and therefore become involved with the tedious details of infrastructure management. Some examples of compute-intensive high-end services that we have developed in our work include numeric solvers, N-body simulators, parallel CFD, stochastic simulation (e.g. monte-carlo), parameter studies, and library-to-library genomic sequence comparison.

However, most high-end applications and services are designed to run in static dedicated environments. It is unrealistic to expect such services to run “out-of-the-box” in a dynamic Grid environment. To meet performance objectives in the Grid, high-end services should be adaptive or *malleable* with respect to system resources. For example, in order to balance resource allocation between competing service requests, resources must be dynamically shared. The system should be able to take resources (e.g. CPUs) away from one request to allow another request to make progress [13].

Implementing adaptivity can be difficult, but we believe that this effort can be amortized since the service will be used repeatedly. In addition, we believe that some resource providers will be willing to host high-end services only if they are adaptive to enable them to give priority to local users. If the high-end service cannot adapt (e.g. release resources) then the resource provider might choose to suspend the service. Conversely, if a hosted service is given only a small amount of resources when it is started, adaptivity could allow it to acquire additional resources later if they become available. We believe these scenarios will be commonplace as Grids evolve.

The AGS has several components. The back-end is the actual service code which is encapsulated by an AGS factory that can clone itself to serve requests. The “back-end” is hosted on one or more ARPs. Hosting means that the ACP has allocated a pool of resources to the service for a negotiated lease period. Hosting decisions are made by the AGS deployer using placement algorithms provided by scheduling middleware [13][21] or specified by the service provider (Figure 2). Once hosted, the AGS factory manages the actual service instances (the AGSIs, that are created by the factory) which carry out service requests for clients. The AGS factory also uses the scheduling middleware to decide how to multiplex the AGSIs across the allocated resource pool.

The AGS back-end supports several interfaces reflecting the exposure of adaptivity (Figure 5). When the AGS factory receives a client binding request, it creates a new AGSI and returns a handle to it. Because resources may come and go dynamically, the AGS factory supports a `notification` interface that allows the AGS to subscribe to events of interest. Notification to the AGS is performed asynchronously. The AGS also implements an `adaptation` interface that allows an ARP to add or remove resources from it. When the AGS factory receives a request on its `adaptation` interface, it decides upon which instances (if any) to act. For example, in the case of an ACP that is managing CPU resources, it could inform the hosted AGS factory that 3 new CPU resources are now available via its `notification` interface. It would be up to the AGS factory to decide if it wanted the resources and what to do with them. The AGSI supports a subset of the AGS interfaces plus service-specific interfaces. The AGSI `adaptation` interface is implemented in a manner specific to the service. For example, adding or removing CPUs from a data parallel service requires data movement for load rebalancing, and this in turn depends on the underlying data structures and other implementation details. On the other hand for distributed services, e.g. a parameter-sweep, adding or removing CPUs is much simpler. The service provider is responsible for implementing these interfaces. The supported events are defined by *EventType*. In the current prototype, these events are limited to ACP-specific events: `NewCPUsAvail`, `CPUsReclaimed`, and `CPULoadChange`. These events also contain specific information about the event. Once the AGS receives an event, it can contact the ACP if necessary (e.g. `NewCPUsAvail` may trigger a request for extra resources). The

```
AGS/AGSI {

factory: // this interface is supported only by the factory (not the instances, AGSIs)
      ServiceType create (); // create AGSI
      void init (LotType); // init AGS with lease info
      void shutdown (); // disable service
      void log_time  (RequestType, TimeType); // logs performance data for a completed request
      PerDataType get_perf_data(TimeFrameType); // perf. data for prior requests over a past time frame

notification:
      void event_occured (EventType); // a subscribed event has occurred

adaptation:

      void new_resource_lot  (LotType); //provide new lot to AGS factory (i.e. adding/removing resources)
      void add_resources (ResourceType); // AGSI - add resources to this AGSI
      void remove_resources (ResourceType); // AGSI - remove resources from this AGSI
      ...
service-specific-interfaces:

      ...
}
```

Figure 5: High-level AGS and AGSI specification

RequestType contains information about a particular request: operation name, parameter values, etc., and *TimeFrameType* represents a past or future time frame, e.g. past hour, last N requests, etc.

3.2 ARP

When high-end services are hosted on the Grid, the need for adaptivity arises due to unpredictable resource sharing both internal to the Grid and external to it (i.e. local non-Grid users). We assume that high-end services are hosted on resources provided by resource providers. The ARP encapsulates Grid resources that are made available to host services (Figure 6).

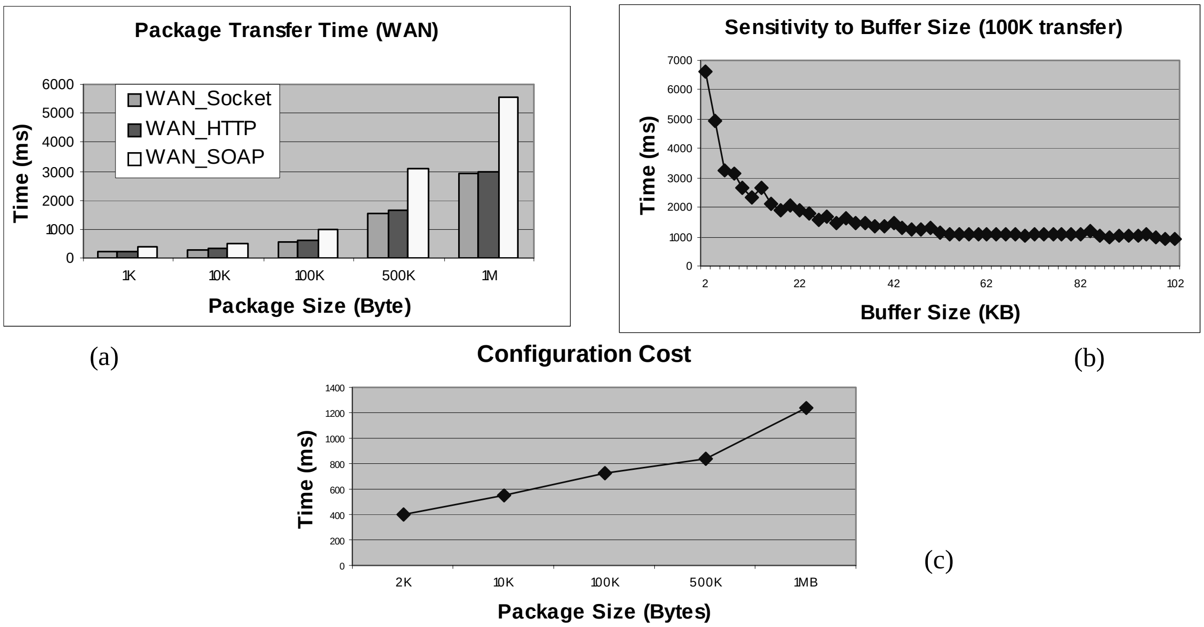
The ARP exposes important internal state via a `query` interface including: the amount of resources it is willing to allocate for a lease period, platform features including hardware/software features, and the resource profile of the contained resources (average number, power, and duration of available CPU resources) over a recent time interval. The *LeaseType* contains the lease duration and the kind of lease (*dedicated*, *shared*, *renewable*). By exposing the leasing model of the resource provider, the service can decide on the degree of performance uncertainty it is willing tolerate. The amount of resources *AmtType* offered by an ARP is specific to the type of ARP. The platform features *PFType* is an extensible type that defines the key features of the platform required by services in order to make hosting decisions, e.g. the kind of process launch capability supported (ssh, MPI, etc.), amount of memory, scratch space available to the service, etc.

```
AdaptiveResourceProvider{
query:
    AmtType avail_amt (LeaseType); // returns amt of avail resources grantable for the desired lease
    LeaseType lease_length(AmtType); // returns maximal lease for the desired amount of resources
    PFType platform_features(AmtType); // returns the platform features of this ARP
    Boolean have_features(PFType); // does the platform have specific features?
    ProfType get_profile (TimeFrameType); // returns perf. profile over past/future time frame

notification:
    void subscribe_event (EventType, AGS); // AGS wishes to subscribe to a particular resource event

allocation:
    LotType alloc(LeaseType, AmtType); // initial pool allocation req. to AGS
    Boolean dealloc (LeaseType, AmtType); // dealloc resources to new amount
    Boolean realloc (LeaseType, AmtType); // increase granted resources to new amount
    Boolean renew_lease ( LotType, LeaseType, AmtType); // renew old lease (in LotType) to new lease

usage: // usage for ACP
    ServiceType deploy (PackageType, LotType); // deploy service package using valid Lot
    void undeploy (ServiceType); // undeploy service from ARP

    ...
}
```

Figure 6: High-level ARP specification

4.0 Results

We have constructed a testbed and deployed our architecture and core services based on GT3 and Tomcat at the University of Minnesota and the University of Virginia connected by I2. Within Minnesota, the services are connected by 100 Mb ethernet. All machines are a mix of Linux and Solaris workstations (2.x GHz Xeon PC with 512 MB running Linux, and Sun Ultra-60 with 1024 MB running SunOS 5.8). This testbed has enabled us to measure basic overheads inherent to our dynamic deployment and invocation architecture. All data presented is the result of five runs with averages shown. Since every aspect of the service is dynamic in our model: installation, deployment, and invocation, it is important that the runtime overheads associated with each of these operations be measured and ultimately mitigated. We first measured the transport portion of the cost of remote service installation (front-end) and deployment (back-end) as a function of the PackageType size. We compared SOAP (using encoded byte arrays) which is the default, HTTP, and TCP/IP (Figure 7a). In general, the SOAP penalty is about a factor of 2 (for WAN transfers). For a PackageType sizes of 100KB, the cost is ~1 sec (WAN) and for 1MB it is ~5.5 sec. If the service is deployed to handle multiple requests (common case) then the overhead of installation/deployment can be amortized. Our conclusion is that for services that do not have extremely large associated datasets, SOAP will be acceptable. We have observed as have others [3] that SOAP performance is very sensitive to SOAP buffer size and a proper choice of buffer sizes is required to obtain acceptable performance (Figure 7b). Tuning these buffers is the subject of future work.



(a)

(b)

(c)

Figure 7: Install/Deployment Transport Cost

Once the service package is delivered to the host site, the service must be configured. This configuration cost includes interaction with the local Tomcat container environment to unpack WAR files, create directories, allocate memory for the service, and start the service (Figure 7c). Configuration was measured on a 2.x GHz Xeon PC with 512 MB running Linux. Similar configuration costs would be paid for both remote deployment and installation since both involve package transmission and interaction with the local container. Configuration costs for packages above 1MB are on the order of seconds. Clearly, deploying the service for each request on demand (transfer and configuration) is expensive (unless the service request is very long running). However, we do not want to rule out such scenarios. The common case of multiple requests will enable the overhead to be easily amortized.

To reduce the cost of dynamic deployment, we next considered several optimizations that can be used in specific situations (Figure 8). For these experiments, we implemented and deployed an eigenvalue solver service that takes an NxN input matrix of 8 byte doubles as its argument. This service has a code size of 15KB which results in a service package of 10KB (when compressed in the WAR format, without GT3 libraries), or 10MB (with all GT3 service libraries). Service tear-down normally involves removal of all traces of the service, including the service package. Instead, it is possible to tear-down the service (removed from Tomcat's memory), but allow the service package to remain "cached" (with a storage cost). By caching we mean that the service package still resides in Tomcat's directory space, but can be re-loaded into a running container later if necessary. The cost of subsequent deployments of the service to this ARP can be greatly reduced as the service package need not be retransmitted (Figure 8a). A second optimization is to shrink the service package to omit redundant system library files. If the ARP is already hosting other Grid services, then it likely has already loaded certain system libraries as part of GT3. These libraries can be loaded as shareable much like shared code segments in OS-level virtual memory. This also applies to service upgrades in which the service package can omit the already delivered and loaded system libraries. This optimization is more powerful as it reduces both the transmission time as well as the configuration cost (Figure 8b).

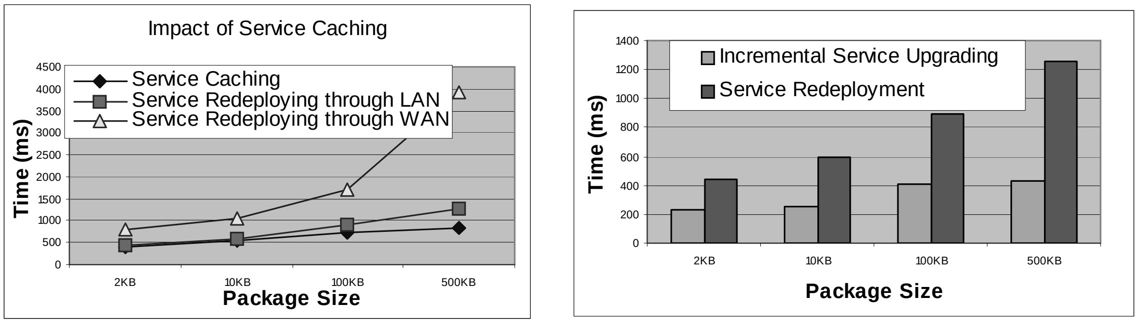


Figure 8: Optimizations. Incremental service package size is 1KB (one class file in the eignenvalue service was changed and only this his class was transmitted).

Once a service is installed and deployed, the next phase of its lifecycle is client access. This is the most important cost from the perspective of the end-user. The end-to-end response time for service access includes the cost of sending the request from the client to the front-end, routing the request to a back-end, and creating a service instance. We have measured this "end-to-end" latency in four scenarios (X-Y) reflecting the network distance from the client to the front-end (X) and from the front-end to the back-end (Y), Figure 9a. The total lifecycle cost in terms of percentages is shown in Figure 9b (shown for the eigenvalue service) with N=113 (yielding a request size of 8*113^2 ~100KB). We show deployment of a full-size service package (10 MB) vs. an optimized one that sends only the service code (10 KB). The importance of incremental deployment is observed as the overhead difference between full deployment (10MB) and incremental (10KB) is about a factor of 4 (80% vs. 20%), (1-4 in Figure 9b). Next, the overhead of service execution once the service is deployed (5-6) is approximately 25%. This cost is largely the transfer of the request input matrix to the front-end and then to the back-end. This cost could be cut in half by sending the inputs directly to the back-end and a description of the inputs (e.g. sizes) only to the front-end to support scheduling. This optimization will be investigated in the future. Finally, a large part of the total cost is building and compressing the service package. Of course, this overhead would likely be amortized over multiple service requests and deployments.

(a)

Scenario	Latency
LAN-LAN	35
LAN-WAN	155
WAN-LAN	148
WAN-WAN	278

(b)

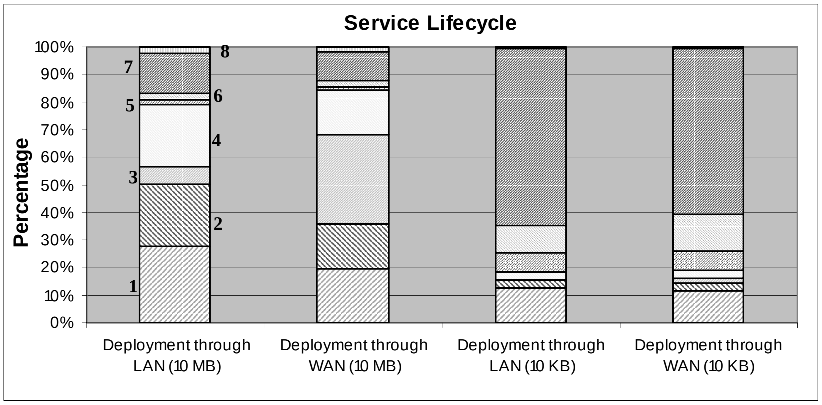


Figure 9: End-to-end Service Cost. Table (a) is latency with no service execution or transmission of input data. Graph (b) is the life-cycle cost and includes: (1) service packaging, (2) front-end installation, (3) package transfer via SOAP, (4) Tomcat configuration, (5) client -> front-end communication, (6) front-end -> back-end communication, (7) AGSI creation and service execution, (8) service tear-down.

5.0 Related Work

Other service based architectures for Grid and network computing have also been proposed such as H_20 [17], NetSolve [2], Ninf [16], CCA/XCAT [1], Service Grid [20], Partitionable Services [10], Soda [11], Sharc [18], and others [12][12][20]. However, few of these service models support the degree of dynamism or adaptation proposed here. In partitionable services, adaptivity is achieved by dynamic service composition at a much coarser grain. In other systems, NetSolve, Ninf, Soda, and others, the service is assumed to be “pre-deployed” and available to support multiple users over some time frame with limited adaptation support. In most cases, adaptation is presumed to be left to the implementation as an optimization. While we also allow the service to persist over some time horizon, we support a dynamic deployment capability based on resource leasing. DynamicTao is a dynamic Corba-based approach that has similar goals, but is not based on Web/Grid services and does not support adaptivity within the service itself. Sharc allows fine-grained adaptation within a single application only. In other systems such as CCA/XCAT and H_20 the service or component instance will be launched on demand when an application requests it. In contrast, our approach allows a service to persist in order to service multiple requests. In a wide-area environment such as the Grid, amortizing the cost of service deployment is critical.

6.0 Summary

We have presented an architecture and prototype implementation for a dynamic Grid service architecture that supports dynamic service hosting - where to host and re-host a service within the Grid in response to service demand and resource fluctuation. Our model defines several new adaptive Grid service classes that support adaptation at multiple levels. In particular, dynamic service deployment allows new services to be added without “taking down” a site for re-configuration, allows the Grid to be much more dynamic, and allows a VO to respond effectively to resource availability and demand. We have also measured the costs of dynamic installation, deployment, and invocation. The preliminary results indicate that the system overhead is tolerable particularly for multiple service requests or when optimizations such as caching or incremental deployment are employed. Future work centers on reducing the basic costs of the infrastructure and the development of middleware components and policies for setting lease lengths, where to host and re-host services, and ARP-level resource management for CPU (ACP) and other adaptive service classes (ASPs).

7.0 Bibliography

[1] R. Bramley et. al, “A Component-Based Services Architecture for Building Distributed Applications,” *Proceedings of the 9th International Symposium on High Performance Distributed Computing*, August 2000.

[2] H. Cassanova and J. Dongarra, “Netsolve: A Network Server for Solving Computational Science Problems,” *International Jour of Supercomputing Applications and High Performance Computing*, Vol. 11, no. 3, 1997.

[3] K. Chiu et al, “Investigating the Limits of SOAP Performance for Scientific Computing,” *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, July 2002.

[4] K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu, “Agreement-based Grid Service Management (OGSI-Agreement),” *Global Grid Forum, GRAAP-WG Author Contribution*, 12 June 2003.

[5] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *International Journal of Supercomputing Applications*, 11(2), 1997.

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

[7] I. Foster, C. Kesselman, J. Nick, S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,” Open Grid Service Infrastructure WG, GGF, June 2002.

[8] Globus GT3: `www.globus.org`, 2004.

[9] A.S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” *Communications of the ACM*, Vol. 40(1), 1997.

[10] A. Ivan et al, “Partitionable services: a framework for seamlessly adapting distributed applications to heterogeneous environments,” *Proc of the 11th IEEE Intl Symp on High Performance Dist Computing*, July 2002.

[11] X. Jiang, D. Xu, “SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms, “*Proc of the 12th IEEE Intl Symposium on High Performance Distributed Computing*, June 2003.

[12] F. Kon et al, “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB,” *IFIP/ACM Intl Conf on Dist Systems Platforms and Open Distributed Processing (Middleware'2000)*.

[13] B. Lee and J.B. Weissman, ``Adaptive Resource Selection for Grid-Enabled Network Services", *2nd IEEE International Symposium on Network Computing and Applications*, April 2003.

[14] M.J. Litzkow et al., “Condor - a hunter of idle workstations,” *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[15] L.F. McGinnis, W. Thigpen, and T. J. Hacker, “Accounting and Accountability for Distributed and Grid Systems,” *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2002.

[16] H. Nakada, M. Sato, and S. Sekiguchi, “Design and Implementation of Ninf: towards a Global Computing Infrastructure,” *Journal of Future Generation Computing Systems, Metacomputing Issue*, 1999.

[17] V. Sunderam, D. Kurzyniec, “Lightweight self-organizing frameworks for metacomputing,” *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, July 2002.

[18] B. Urgaonkar and P. Shenoy, “Sharc: Managing CPU and Network Bandwidth in Shared Clusters,” *IEEE Transactions on Parallel and Distributed Systems*, January 2004, Vol. 15, No. 1.

[19] G. Wasson, N. Beekwilder, M. Morgan, and M. Humphrey, “OGSI.NET: OGSI-compliance on the .NET Framework,” *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.

[20] J.B. Weissman and B. Lee, “The Service Grid: Supporting Scalable Heterogeneous Services in Wide-Area Networks,” *2001 Symposium on Applications and the Internet*, January 2001.

[21] J.B. Weissman, D. England, and L.R. Abburi, “Integrated Scheduling: The Best of Both Worlds,” *Journal of Parallel and Distributed Computing*, 63(6), June 2003.

[22] R. Wolski, et al. “G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid,” *Proceedings of the 2001 International Parallel and Distributed Processing Symposium*, 2001.