A Framework for Dynamic Service Adaptation in the Grid: Next Generation Software Program Progress Report

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

Abstract

This project addresses the problem of dynamic service adaptation in the Grid. The need for adaptation arises due to both resource and service demand uncertainty. With NGS support, we are tackling several key issues in this problem space: (1) a dynamic OGSA-based Grid service architecture to support dynamic service hosting - where to host and re-host a service within the Grid in response to service demand and resource fluctuation [13]*, (2) resource management middleware that dynamically decides how many resources to allocate to a request and where a request should run* [10][12]*, (3) a dynamic leasing framework that decides how many resources to lease to a service to handle future requests* [2][3]*, (4) and a new model of service robustness that can describe the sensitivity of a service to Grid fluctuations* [4]*. In this short report, we present material on (1) and (3) in the interest of brevity. Papers describing all aspects of the project are contained in the bibliography.*

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] and Legion [9]. 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. 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 - where to host and re-host a service within the Grid. We have also developed 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".

2.0 Dynamic Grid 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 [11], etc. Grid system services provide core functionality that is required by application-level Grid services.

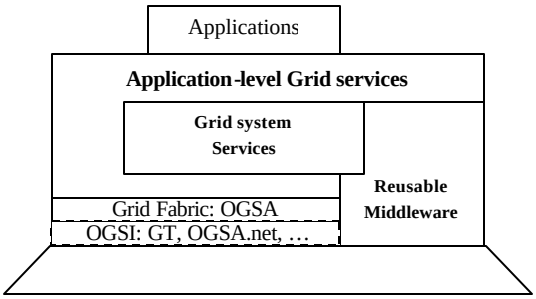


Figure 1: Grid Stack. The bold boxes are addressed in our system

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. 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 Grid service AGS model is attractive for several 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.

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 configured 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 [1].

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.

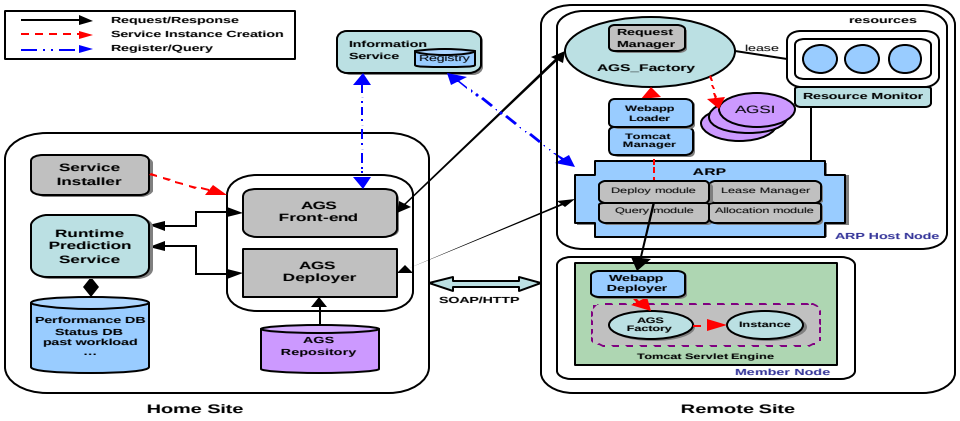


Figure 2: Dynamic Service Architecture

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 dynamic deployment lifecycle is shown below (Figure 3). 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. APIs for all components in Figure 2 can be found here [13].

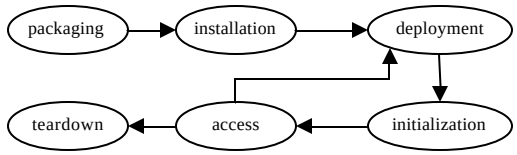
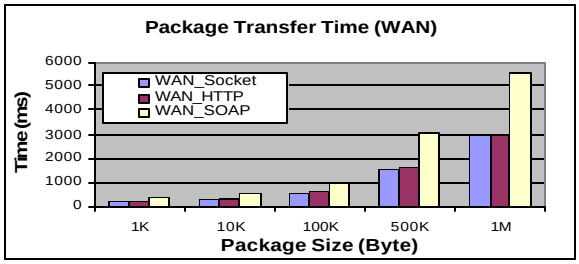


Figure 3: Service Lifecycle. All elements of the service lifecycle are supported in the architecture. Note that dynamic service access may induce additional back-end deployments.

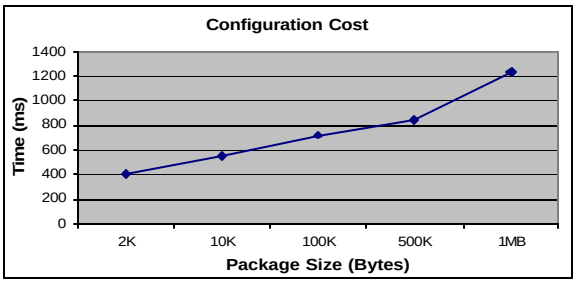
2.1 Results

We have constructed a testbed and deployed our architecture to examine its costs and benefits. 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 4a). 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.

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 4b). 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 5). 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 5a). 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.



(a)

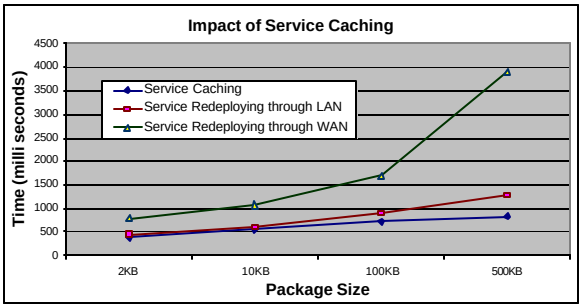


(b)

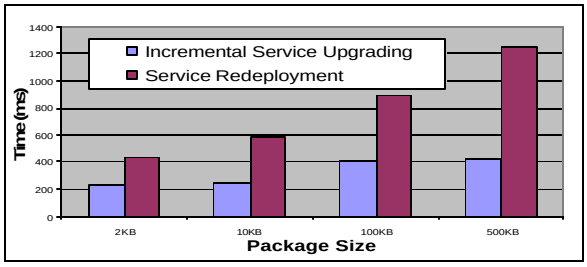
Figure 4: Install/Deployment Transport Cost

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 5b).



(a)



(b)

Figure 5: Optimizations. Incremental service package size is 1KB (one class file in the eignenvalue service was changed and only this his class was transmitted).

3.0 Dynamic Service Leasing

When a service provider of an AGS wishes to host on an ARP, it leases resources from the provider. We assume that there is a cost for leasing as the ARP resources may be provided by a third party, e.g. Sun's utility computing model in which cycles were recently sold on e-bay. We consider the resource leasing problem from the perspective of the service provider. The nature of the problem is such that the demand for the service and the processing times of individual requests are unknown, but can be modelled with appropriate probability distributions. Still, the service provider is faced with the conflicting goals of leasing enough computational resources to provide an adequate level of service and keeping the cost of leasing to a minimum. Leasing too many resources to host a service incurs unnecessary cost. However, leasing too few resources results in long wait times and client dissatisfaction. In our model we place a cost on the average wait time of client requests. Such a cost can represent a loss in revenue or a loss in goodwill. Due to its stochastic nature we model and solve the resource leasing problem in a dynamic programming framework. Given the cost structure of the lease arrangement, the resulting policy provides state-dependent rules for service providers to acquire on-demand resources and to terminate those leases when it is cost-effective to do so. We model cost using three terms, an initial acquisition cost to acquire the resource at the beginning of a time period, a hold cost that is linear in the time a resource is held, and a dynamic acquisition cost that occurs due to under-provisioning between time periods. A full treatment of our mathematical formulation of this problem is outside the scope of this report, but can be found in [3].

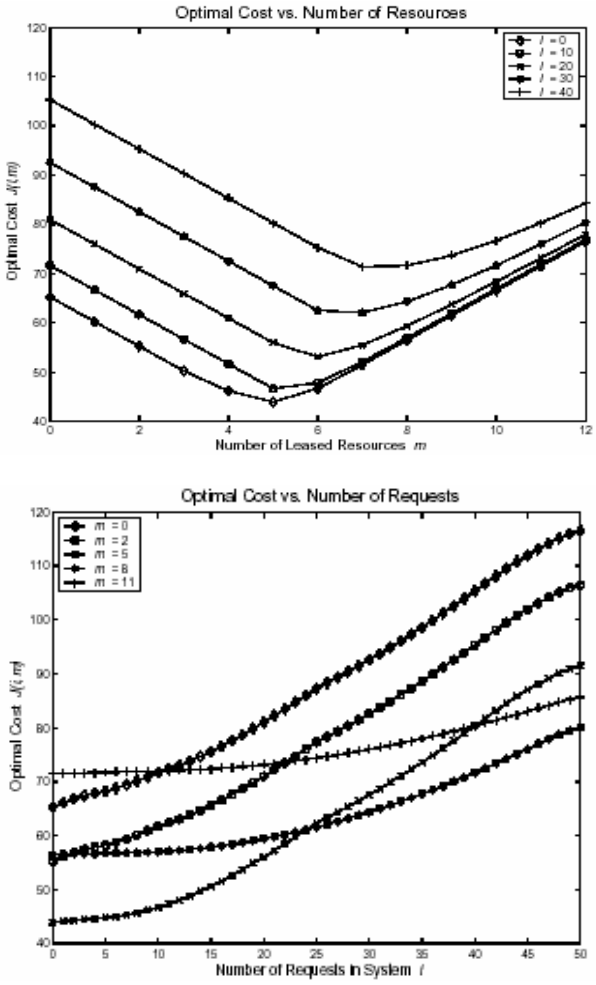


Figure 6: Leasing cost vs. number of resources and requests

We characterize the optimal cost function and we find that the cost is more sensitive to leasing too few resources than to leasing too many resources. This indicates that it is better for a service to be slightly overdeployed than under-deployed (Figure 6) as

evidenced by the spread of cost function at the left end of the graph and the closeness at the right end of the graph. This indicates that when demand for the service is known to exist but is highly irregular, it is better to be slightly over-deployed than under-deployed. Knowledge of the sensitivity is also beneficial in the case where consistency and predictability of the leasing cost is of concern, e.g. service providers who budget for leasing costs. Regarding cost versus number of requests in the system, we show the optimal cost for five different levels of m (the number of resources.) We see that the cost is nondecreasing in the number of requests. The lowest costs are naturally achieved when there are very few requests in the system since there is no waiting cost. We also show that the optimal cost function is not very sensitive to the number of requests in the system. This can perhaps be taken as good news since, unlike the number of leased resources, the number of requests is random and cannot be directly controlled.

4.0 Summary

This project is addressing fundamental issues relating to how services can cope with the dynamics of the Grid. We presented an architecture and prototype implementation for a dynamic Grid service infrastructure 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. The costs associated with dynamic deployment were shown to be tolerable. We also presented a dynamic leasing strategy whereby a Grid service provider can optimally decide how many resources to lease from a provider given knowledge of cost parameters and general statistics of the service demand profile. This work is currently being implemented in our middleware. Prior work not presented here focused on reusable middleware for resource management at several levels - deciding where to ship a service request (if there were multiple deployments), deciding when to replicate a service if demand was high, and deciding how many resources to allocate to a specific service request from the pool leased to the service provider. Our current work is focusing on adaptation to support reliable and robust Grid services that can withstand extreme perturbation due to resource failure or removal or extremely high service load. Our future work is to develop an integrated service infrastructure for handling extreme events for data-intensive services that exploits the best properties of Grid, P2P, and commodity cluster service models.

5.0 Bibliography

[1] 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.

[2] D. England and J.B. Weissman, "A Stochastic Control Model for the Deployment of Dynamic Grid Services," *5th IEEE/ACM International Workshop on Grid Computing* 2004.

[3] D. England and J.B. Weissman, "A Resource Leasing Policy for On-Demand Computing," invited to the I*nternational Journal of High Performance Computing and Application*s (IJHPCA) 2005.

[4] D. England and J.B. Weissman, "A New Metric for Robustness with Application to Network Services," University of Minnesota Computer Science Technical Report, 2005.

[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] 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.

[11] 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.

[12] 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.

[13] J.B. Weissman, S. Kim, and D. England, "Supporting the Dynamic Grid Service Lifecycle," to appear in *CCGrid* 2005.