ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-262

The Taming of the Grid: Virtual Application Services

by

Katarzyna Keahey and Khalid Motawi
{keahey, kmotawi}@mcs.anl.gov

Mathematics and Computer Science Division

Technical Memorandum No. 262

May 2003

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, SciDAC Program, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Disclaimer

Available electronically at http://www.doe.gov/bridge

Available for a processing fee to U.S. Department of Energy and its contractors, in paper, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Contents

The Taming of the Grid:
Virtual Application Services

by

Katarzyna Keahey and Khalid Motawi

{keahey, kmotawi}@mcs.anl.gov

Abstract

In this report we develop a view of the Grid based on the application service provider (ASP) model. This view enables the user to see the Grid as a collection of application services that can be published, discovered, and accessed in a relatively straightforward manner, hiding much of the complexity involved in using computational Grids and thus making it simpler and more accessible to a wider range of users. However, in order to satisfy the requirements of real-time scientific application clients, we combine the ASP model with representation of quality of service about the execution of services and the results they produce. Specifically, we focus on real-time, deadline-bound execution as the quality of service derived by a client. We describe an architecture implementing these ideas and the role of client and server in the context of the functionality we develop. We also describe preliminary experiments using an equilibrium fitting application for magnetic fusion in our architecture.

1 Introduction

In the prevalent model of software sharing, service providers typically port their software to a standard set of platforms; community users then install and use this software on their machines. This process is often arduous from the viewpoint of both the user and the provider. The user must go through the (usually complex) process of installing the code and its dependencies, then must maintain that code, and also must update the installation whenever a new version comes out. This process is made more difficult by the fact that scientific codes are often updated frequently to reflect improvements in modeling techniques. From the provider's point of view, the necessity of supporting the code on even a limited set of platforms can require significant cost and effort. In addition, maintaining and debugging a community code on an unfamiliar platform can mean that a significant amount of effort is spent simply in reproducing, let alone fixing, a problem.

The emergence of computational Grids [1] motivated some communities, such as the National Fusion Collaboratory (NFC) [2], to adopt the Application Service Provider (ASP) model [3] so that code is shared by members of a virtual organization (VO) through remotely accessible "network services". In this model, code providers maintain the code on a familiar and easily accessible set of platforms and make that code available to remote users belonging to the VO. While a service interface allows a code to be published, discovered, and accessed much like a Web page, the demands placed on service execution can be considerably larger. For example, some of the NFC code executions must satisfy strident real-time constraints.

Thus, it is important that application servers be provided with the capability to negotiate and provide quality-of-service (QoS) guarantees required by a user. Frequently, these QoS guarantees will depend on the underlying resource management framework; hence, the application server will have to negotiate with other resources, such as hardware or software license providers. To support this capability we require a general-purpose infrastructure capable of automatically and dynamically establishing and managing relationships between multiple resources. The user will no longer map specific binary installations, application configurations, and licenses to a hardware platform in order to satisfy his or her QoS objectives; this generic functionality should be provided by an infrastructure. Much like a virtual storage system might be implemented by combining the capability of several actual storage systems, these *virtual application services* (VAS) will draw on a variety of application configurations, resources and management capabilities. This view of computational Grids hides much of their complexity, allowing it to appeal to a broader range of users.

The VAS model requires the development of new capabilities. One of these is the ability to specify application-level QoS interfaces. Since the user is no longer dealing directly with hardware, but rather with application-specific qualities, specifying expectation about service execution in terms of the hardware constraints (e.g., "use 8 processors") is no longer relevant. Furthermore, since the hardware resources are no longer well known to the user, their effect on the application is best estimated and described by the service provider. The user should be able to specify constraints on service execution in application-specific qualities (such as time bounds, application-specific accuracy measures, and cost) that can be used to map application constraints to a given machine. Bridging the gap between resource specific descriptions developed by the service provider, application models, and user requirements requires the development of models and protocols that allow the service provider to automatically map user demands into the available resources and make estimates and reservations satisfying these demands.

In this paper we describe the concept of virtual application services and architecture for QoS-based application service execution that realizes the concept of network services. We present an implementation of this architecture based on the Open Grid Service Infrastructure (OGSI) model [4, 5] as implemented by the Globus Toolkit®. In addition, we discuss the results obtained by experimenting with one of our target applications with execution time as our QoS metric. Although our approach was motivated by a scenario using code resident on a set of machines, we believe that the resulting architecture can

also be used in a scenario where a VO can broker highly portable codes to a set of resources that can be provisioned to run those codes.

This paper is organized as follows. In Section 2 we introduce the scenario that motivated our work, and we derive our problem definition. In Section 3 we describe the architecture underlying our solution, and in Section 4 we discuss the architecture implementation. In Section 5 we present results of our work as applied to our motivating scenario. In Section 6 we describe related work, and in Section 7 we conclude with a brief summary and an outline of future work.

2 Requirements and Scenarios

We begin with an example scenario motivating our work. Based on this scenario, we then define the problem we address in this paper.

2.1 Motivating Scenario

Our motivating scenario comes from the National Fusion Collaboratory (NFC) project [6], which defines a virtual organization devoted to fusion research and addresses the needs of codes running during fusion experiments. Magnetic fusion experiments operate in a pulsed mode producing plasmas of up to 10 seconds duration every 15 to 20 minutes, with multiple pulses per experiment. Decisions for changes to the next plasma pulse are made by analyzing measurements from the previous plasma pulse (hundreds of megabytes of data) within roughly 15 minutes between pulses. This mode of operation could be made more efficient by the ability to do more analysis and simulation in a short time using codes running on remote resources if only their execution time could be guaranteed.

The specific applications targeted by our work are EFIT [7], an analysis code computing magnetic equilibrium reconstruction, and TRANSP [8], a code computing particle transport. A typical scenario for a run is as follows. A scientist at one of the NFC sites (a client site) needs to remotely run code installed and maintained at another NFC site (a service provider site) during an experiment within time bound T. Before the experiment, an automated script is prepared that will download experimental data for the application input once that data becomes available. Since some applications, such as TRANSP, can run for a long time, a suitable "short-running" configuration, capable of executing within T , is prepared by the service provider. To ensure that the code executes with the required QoS (in this case: within time T), the scientist at the client site makes a reservation with the application server and as a result is guaranteed code execution within T any time it is requested during the experimental window (roughly a day). Since only a few such executions may be requested during that day, and the service provider resources have to be shared with other clients, it is essential that resource allocations not be overgenerous and that other codes can share the resource with the time-critical application, getting preempted whenever the situation requires.

In other words, the reservation made with service provider must not preclude other computation on a resource. Hence, the time-critical runs must be able to preempt all other computations and claim as much CPU power as is needed. After the high-priority experimental run completes, other processes may reclaim their CPU share.

2.2 Problem Definition

The following assumptions describe our environment from the *service provider's* point of view. For the purposes of this work we assume that we have a *deployment domain* for our application service: a set of preconfigured application installations on a set of dedicated resources. We can make resource reservations and enforce priority-based preemption on those resources. We further assume that we have multiple application configurations to choose from, each described by *metadata* referring not only to installation information about a particular application but also to experimental data in the form of, for example, the application execution times and application-specific quality measures on the results they produce. This data is defined by the service provider as part of the process of service installation. In this particular work we also assume that we have complete control and consequently full knowledge of this domain. These assumptions can later be relaxed to incorporate Grid monitoring, adaptive scheduling and resource management techniques, and movement of portable code. We introduce them here to provide a tractable solution for the problem described above.

From the *service client's* point of view we must address two challenges:

- Provide a persistent, remotely accessible application server that can create application services on demand, scaling in the number of clients and thus freeing the client from the necessity of installing and maintaining an unfamiliar code.
- Allow the clients to specify execution constraints in terms of *application-specific qualities* rather than resource-specific qualities, and later enforce those constraints, thus ensuring QoS in a resource environment that is no longer familiar to, or controlled by, the client.

Providing reliable QoS entails monitoring the execution of the selected service instance, potentially restarting it, and so forth. These actions should be transparent to the client; that is, we require an intermediary to represent the service instance to the client. We use the term *virtual application service* to describe the service that fulfills these creation and execution constraints.

3 Architecture

In this section we describe the architecture designed to implement the scenario described in the preceding section, and we identify the role of each component within that architecture (the implementation details of our prototype are described in the following section.)

The architecture is depicted in Figure 1. The client interacts with this architecture in the following steps:

1. **Client Negotiation**. The client first negotiates with the VAS factory for the future creation of a service with certain QoS guarantees on its functionality. After successful negotiation the factory issues a service level agreement (SLA) that describes the agreed upon QoS constraints to be available in a certain timeframe (possibly immediate timeframe). Although in general an SLA can be very complex and involve complex information related to change, payment, and security issues [9], in our current prototype it is a simple statement guaranteeing certain qualities to be fulfilled.

2. **Service Instantiation**. Service instantiation takes place when the client submits the SLA to the VAS factory during the agreed upon availability window. Service instantiation involves creating a service proxy (or a virtual service) that represents the service to the client. A proxy handle is then returned to the client.

3. **Service Execution.** The client requests the execution of desired actions through the proxy. Using the proxy handle, the client can also obtain information related to the execution of these actions as well as manage and terminate the service.

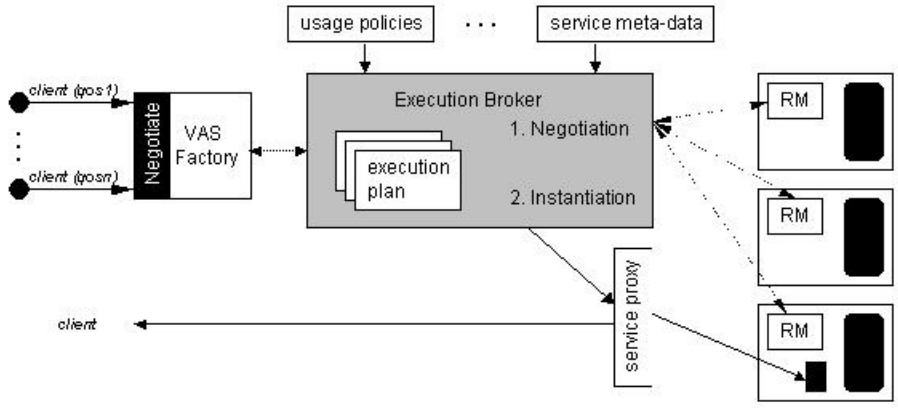

Figure 1: VAS architecture: The key component is the execution broker, which interacts with resource managers (RM) associated with each of the resources relevant to service execution. The rectangles to the right represent the resources with possibly special software installed on them, and the oblong black rectangles a set of service installations available on a given resource. A black rectangle represents the execution of a specific application service; the client interacts with that service through the service proxy.

We next describe in detail each of the architecture components. The key functionality is provided by the execution broker (EB) that implements the main functions of the VAS factory (that of reserving and claiming resources). In order to do that, the execution broker interacts with resource managers (RMs) specific to every resource that might be necessary for service execution.

3.1 VAS Factory

The VAS factory is a persistent service that extends the concept of OGSA factory [4] by a negotiation interface. It thus fulfills two roles: that of a negotiation agent and that of an instantiation interface. In its role as a negotiation agent, it creates service level agreements for execution of actions associated with specific services (similar to task service level agreements described in [10]). Because its function is restricted to service creation and because negotiation and instantiation are assumed to be relatively short operations in our model, it is assumed that the VAS factory can handle many clients with little delay.

3.2 Execution Broker

The execution broker implements much of the functionality of the VAS factory. Its main tasks are building an execution plan capable of running a service with the quality of service requested by the client, and then implementing that architecture when the service is instantiated.

3.2.1 Designing an Execution Plan

In designing an execution plan, the execution broker searches for the best ways to satisfy the QoS requested by the user, given available resources and application installations.

The capabilities of a specific application installation are evaluated through *application metadata* describing its configuration. These metadata include both system-related information, such as for example installation and environment details of a specific executable, and application-related modeling information that allows the execution broker to estimate how fast a given application can be executed on available resources. The latter may be provided in terms of scalability or other analytical data or may be based on empirical data of previous executions.

After hardware needs have been determined based on the metadata, the availability of hardware resources is determined through interaction with resource managers. During this process the execution broker communicates with resource managers running on each machine belonging to the deployment domain of the application, evaluates their availability, and makes reservations. Although in this work we focus on CPU reservation, in general the execution plan may comprise multiple resource reservations (CPU, network, disk). It may also involve pre-execution operations (data prefetching), starting up and combining multiple services, staging data, and the like. For such complex execution plans, it is important to define QoS milestones to aid in monitoring execution.

Hardware and software availability must be further reconciled with factors such as use policies specified by a virtual organization and resource owner. A user may, for example, have suitable execution privileges only for a certain service configuration or only on a

certain set of platforms or may face QoS constraints (such as priority execution). We partially address this problem elsewhere [11] but have not yet integrated the solution in this work. Effectively, in the planning stage the execution broker binds the requested actions to specific resource managers for future execution.

The result of this phase is an SLA, forwarded to the client, and an execution plan maintained by the execution broker that can be retrieved and executed based on the SLA.

3.2.2 Service Instantiation

In the service instantiation phase, the execution broker retrieves an execution plan based on an SLA presented by the client. It then implements the execution plan by claiming all the reservations and configuring and launching its components. In this phase, the execution broker works with resource managers to ensure that the launched application processes are given resource allocations estimated when the execution plans were formed, that they are load-balanced over the deployment domain, or that they are replicated in order to improve reliability of execution. After instantiating the service, the execution broker creates a service proxy and returns it to the client.

3.3 Service Proxy (Virtual Service)

The proxy abstracts the notion of the actual service execution. For example, in order to improve the reliability of execution, the actual service execution may be replicated over the Grid resources. Also, the proxy may monitor QoS milestones of an execution, send updates to the client, and adaptively readjust it as need arises in order to meet the QoS.

3.4 Resource Manager

During the negotiation process the execution broker communicates with resource managers associated with each resource belonging to the deployment domain of the application. In the current design the main purpose of resource managers is to keep track of reservations for a given machine. The RMs also provide an interface to resource-specific mechanisms implementing, for example, priority assignments and performing some monitoring functions.

4 Implementation

We implemented a prototype of the architecture described in the preceding section, using the technology preview implementation of OGSI [5] provided by the Globus Project™. We extended the OGSA abstraction of service factory by mechanisms allowing the user to negotiate service creation with specific QoS constraints. We also relied on OGSA discovery mechanisms for publishing and discovering services. The enforcement of CPU reservations was implemented by using the Dynamic Soft Real-Time (DSRT) scheduler [12] described in Section 4.4. The main components of the architecture were implemented in Java and made available as Grid services; they are described in the rest of

this section. In our implementation we assumed that we have one factory and one execution broker per application (or service type).

4.1 VAS Factory

The VAS factory is a persistent service and a front-end component of the execution broker. In the current implementation the client negotiates with the VAS factory based on two constraints: execution-time bounds (with limited accuracy) and accuracy (with unlimited execution time). These constraints are closely tied to specific installation configurations and affect execution plan decisions made by the execution broker, as well as application-related factors such as the number of iterations made by the network service.

The negotiation is conducted in two phases: an iterative negotiation phase, where the desired QoS is agreed on, and a reservation phase, which yields a service level agreement. In the negotiation message a user requests a reservation for execution such that, if it is submitted within a certain availability window, it will fulfill certain QoS requirements. The QoS measures we currently use are execution time and (as a crude measure of accuracy) the number of timesteps by which the computation advanced. These lead to a QoS request of the following form:

```
<complexType name=QOSType">
  <sequence>
    <element name="schedule"/>
      <complexType name="eb-types:ScheduleType">
        <sequence>
          <element name="startTime" type="dateTime"/>
          <element name="windowSize" type="int"/>
        </sequence>
      </complexType>
    </element>
    <element name="measures"/>
      <complexType name="eb-types:MeasureType"/>
        <sequence>
          <element name="timeBound" type="int"/>
          <element name="timeSteps" type="int"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

The factory replies with a list of proposed SLAs and a flag indicating success of the operation. If the operation succeeds, the SLA list contains only one SLA element fulfilling the requested QoS; if not, the list contains other proposed SLAs (counteroffers). The user can then choose an SLA and make a reservation, or can continue the negotiation process. The SLA element is as shown below; in addition to the QoS it underwrites, it contains a handle and an expiration time on that handle. The expiration time on the offers is set so that the time bound measure fits in the execution time window.

```
<complexType name="SLAType">
   <sequence>
     <element name="handle" type="string"/>
     <element name="expirationTime" type="dateTime"/>
     <element name="qosReqs" type="eb-types:QosType"/>
   </sequence>
 </complexType>
```

The original expiration time of counteroffers is short; the reservation confirmation extends it. After the client reserves the selected SLA, the VAS factory confirms it and releases any other reservations. In the current implementation the SLA agreements are not signed; we assume a trusted environment.

This negotiation protocol is similar to SNAP [10], differing in that in case of failure to obtain the exact requested SLA, the negotiation process returns multiple SLAs instead of just one. Although this requires an additional confirmation message, it also increases the probability that an acceptable SLA is returned and therefore stands to shorten the negotiation process.

4.2 Execution Broker

At startup, the execution broker configures itself to so that it can create a particular application by reading metadata service descriptions associated with that application instance. The application metadata contains both *configuration metadata* (information, describing specific application configuration on a given host) and *QoS metadata* (information relevant to estimating QoS properties of each installation). The configuration metadata contains information about the required environment setup, the location and activation information about the executable on a given machine, additional input arguments, pre- and postprocessing scripts, and similar configuration details. The QoS metadata currently contains the number of timesteps for a given configuration as well as execution time, $t_{100\%}$, measured while running the application at full application allocation of CPU on a given machine (supplied by application provider at configuration time).

The QoS metadata is used to estimate the execution time for the application configuration given the available resources. Given the execution time $t_{100\%}$ and the requested execution timebound ($t_{requested} < t_{100\%}$), we use a simple function ($allocation\% = (t_{100\%} * 100\%) / t_{requested}$) to estimate the percentage of CPU that needs to be allocated in order to satisfy the request. The execution broker then places a reservation for the CPU allocation at a time indicated in the client's request with the resource manager.

If the reservation succeeds, the execution broker formulates an execution plan that involves starting up the application as described in the configuration metadata and then claiming and assigning the CPU reservation through the resource manager. Having obtained the resource, the execution broker now issues an SLA guaranteeing application execution within the requested time constraint. When the SLA is claimed, the execution

broker puts the execution plan into action, creates a service proxy, and returns the proxy handle to the client.

4.3 Resource Manager

The resource manager is based on a similar construct introduced in GARA [13]. Because the execution broker may make tentative reservations during the negotiation process, the resource manager supports a two-stage reserve/confirm reservation process. Reservations are associated with an expiration time; if the reservation is not confirmed before the time expires, the reservation is deleted.

The main tool used by the resource manager is a slot table. It provides the general capability to reserve a number of units of a resource for a particular range of time. The RM uses the slot table to store the percentage of CPU that has been allocated to each reservation. The slot table also supports the ability to search for available units based on two criteria: best allocation within a specified time range, and earliest time range for a specified number of units. In response to a reservation request, the RM creates a slot. A slot identifier is then returned to the execution broker.

Reservations are claimed by providing the RM with a slot identifier and the process identifier of a running process. The RM uses slot start and end times to create timed events that instruct DSRT to start and stop managing the CPU allocation of a process. Processes that do not complete during the span of their reservation will continue to run, but without special allocations.

4.4 DSRT Scheduler

The Dynamic Soft Real-Time scheduler provides the basic capability to reserve a percentage of the CPU for a given process. It is also preemptive; that is, a process that does not hold a CPU allocation from DSRT can be deprived of its CPU share by a process that does hold such a reservation. Typically, at least 10% of the CPU is left unreserved to ensure that important nonreserved systems processes are not starved out. In our implementation we increased this number to 30% to account for the CPU time consumed by OGSA implementation. Thus, in our implementation we have a system allocation of 30% and an application allocation of 70%.

To provide a CPU allocation, DSRT sets the priority of a process to the highest level priority for a percentage of time in a period corresponding to the percentage of CPU reserved. For example, a process with a 50% reservation will have the highest priority for 50 out of 100 milliseconds.

4.5 Service Proxy

The service proxy presents the service interface to the client, hiding the fact that the actual service may be replicated in order to provide better reliability of execution. It implements the service interface by forwarding the requests on to an actual service implementation. The service proxy is created by the execution broker when a client presents a valid SLA. In addition to duplicating the interface of the service itself, the proxy supports additional operations that are used by the execution broker. These operations include setting a reservation handle and binding the proxy to the actual service.

4.6 Service Instance Implementation

Since our target scientific applications are implemented as C programs and since C binding is not yet available in OGSA, our service instances were represented as a combination of the OGSA job manager (JM) service, a C program wrapper, and the application. We use the C program wrapper to access application-specific information such as the process id. The application is activated by using the JM to launch a C wrapper, which claims the reservation slot by sending its process identifier and a reservation handle to the resource manager. The wrapper then replaces itself with the application executable. Since each OGSA JM instance can manage exactly one process at a time, an instance of the combined service is created for each application request.

The availability of a C hosting environment for OGSA would enable us to simplify this process by associating with a given application (now implemented as an OGSA service) service data elements representing application-specific information such as the process id. This information would be then accessible by invoking the FindServiceData operation standard for all Grid services.

5 Empirical Results

We ran preliminary experiments in order to evaluate how well our implementation estimates the execution time, makes corresponding resource reservations, and then, based on those reservations, delivers results within the estimated time. In our tests, we used the EFIT application described in Section 2 as representative of the class of applications that normally run in these conditions. To simplify the problem, we assumed the same set of fitting parameters for each run. Since one of the premises of our work is that any CPU allocation not claimed as a reservation may be used by non-time-critical executions, we verified our results on a loaded system, that is, with other time-shared (not scheduled through DSRT) and real-time (scheduled through DSRT for the remainder of application allocation) processes present during a time-critical execution.

For each test, we repeated the following sequence. A client requests an EFIT execution, fixed in the number of timesteps (the same in all tests), within a certain time bound and save the resulting SLA. A load of real-time and time-shared jobs are started, and subsequently the client claims the reservation. As a result, EFIT is started by the execution broker, and its execution time is recorded. After the EFIT finishes, the background job load is shut down. To test the behavior of the system for different CPU reservations, we varied the timeframe to correspond to a percentage of CPU allocation for a fixed number of timesteps.

Figure 2 compares predicted ("promised") execution time (based on formula in Section 4.2) and actual (measured) execution times. The results show relatively high variability for low CPU allocations (10% and 20%) and little variability for larger CPU allocations; in the worst case the standard deviation across CPU allocations was 10%. The variability was much greater for lower CPU percentage ranges because the increased swapping between active and idle status for the application over longer execution timeframes caused greater numbers of interruptions and disk or cache effects.

To assess the influence of different kinds of job loads on a time-critical run, we compared the results of a time-critical EFIT execution under an unloaded system, a system loaded with time-shared jobs, and a system loaded with real-time. Figure 3 shows median values of obtained results. The behavior of EFIT is similar under all three conditions, showing that background jobs do not seriously impair the effectiveness of our system. Also, the patterns of result variability in the loaded experiments were the same as in the unloaded experiments.

Overall, we conclude that this system has potential to provide QoS for deadline-bound execution. The 10% variability in execution time is acceptable because EFIT operates under soft real-time conditions that can tolerate this amount of variability. Although our predictions can be made more conservative based on this information, we prefer the approach where we attach the variability as a "confidence measure" to the SLA, and leave this decision to the user. Another observation that allowed us to improve our initial handling of predictions and reservations is that because of the relatively high variability, we decided not to based time-critical executions on CPU allocations of less than 30%.

We recognize that our method depends on the availability of historical data for pre-configured executions (as those become available during the preparations for an experiment). Prediction in different situations has been addressed by others [14]; however, the same principles apply: a measure of confidence, reflected in the SLA, could then be attached to those predictions.

6 Related Work

Different variations of the ASP model have been explored and found useful in the Grid context before [15-18]. The work of [18] is especially remarkable in that it dynamically addresses the problem of service performance scalability, which is relevant from the

perspective of establishing QoS-based contracts between the service provider and the client.

At the same time, the idea took hold that QoS representations can be made not only about underlying resource-level elements, but also about object and components [19, 20]. We combine these ideas with the resource management technology available for the Grids, specifically the work on resource reservation developed in GARA [13], and show how it can be leveraged in providing real-time QoS for applications.

Other related work, such as combining QoS-aware aggregation models [21] with workflow models [15] can be used to extend this work to include end-to-end QoS for multicomponent applications..

7 Conclusions and Future Work

We have presented an architecture enabling a client to enter into a QoS contract with an application service provider. The application service provider can then perform provisioning actions that enable it to execute on a chosen resource within the QoS requested by the client. This architecture allows us to bridge the gap between resource-specific descriptions developed by the service provider and user requirements, allowing the service provider to automatically map user demands into the available resources and make estimates and reservations realizing their goals. We also showed how this approach can also be used to satisfy real-time quality-of-service requirements in real-life scenarios. In order to satisfy them fully, more complex relationships between resources may need to be scheduled; however, in this paper we provide a proof of concept of the feasibility of this approach.

In order to provide end-to-end QoS, the execution broker execution plans will be more complex and involve more complex multiresource provisioning. Furthermore, realistic multi-user scenarios will require combining the provisioning with use policies enforcement to ensure that codes and resources are used as intended. We introduced the service proxy component to create potential for application-level reliability and adaptivity but have not realized it in the current implementation. Improved strategies (such as replication), as well as adaptive techniques, especially combined with execution milestone monitoring, will allow us to demonstrate the potential of this component. Moreover, combining work on application prediction and SLAs will make this work applicable to a wider range of applications.

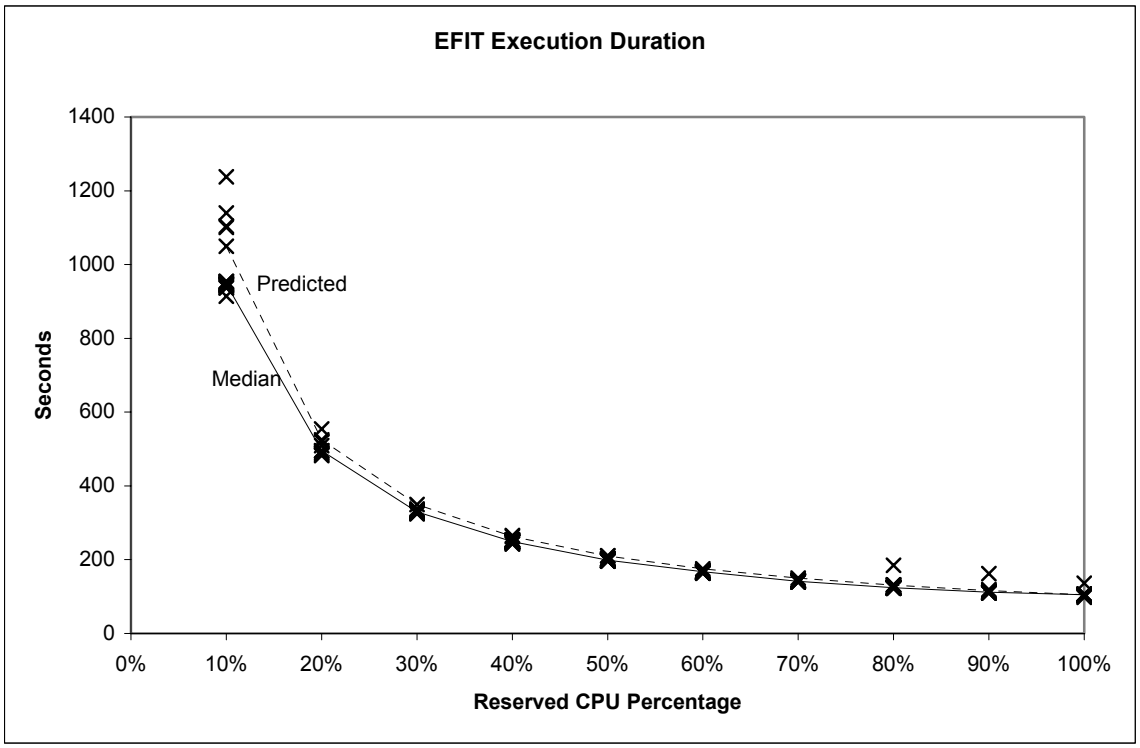

Figure 2: Execution time per CPU allocation. The graph shows relatively high variability for low CPU allocations (10% and 20%) and low variability for higher CPU allocations. The data is based on 15 measurements per CPU allocation.

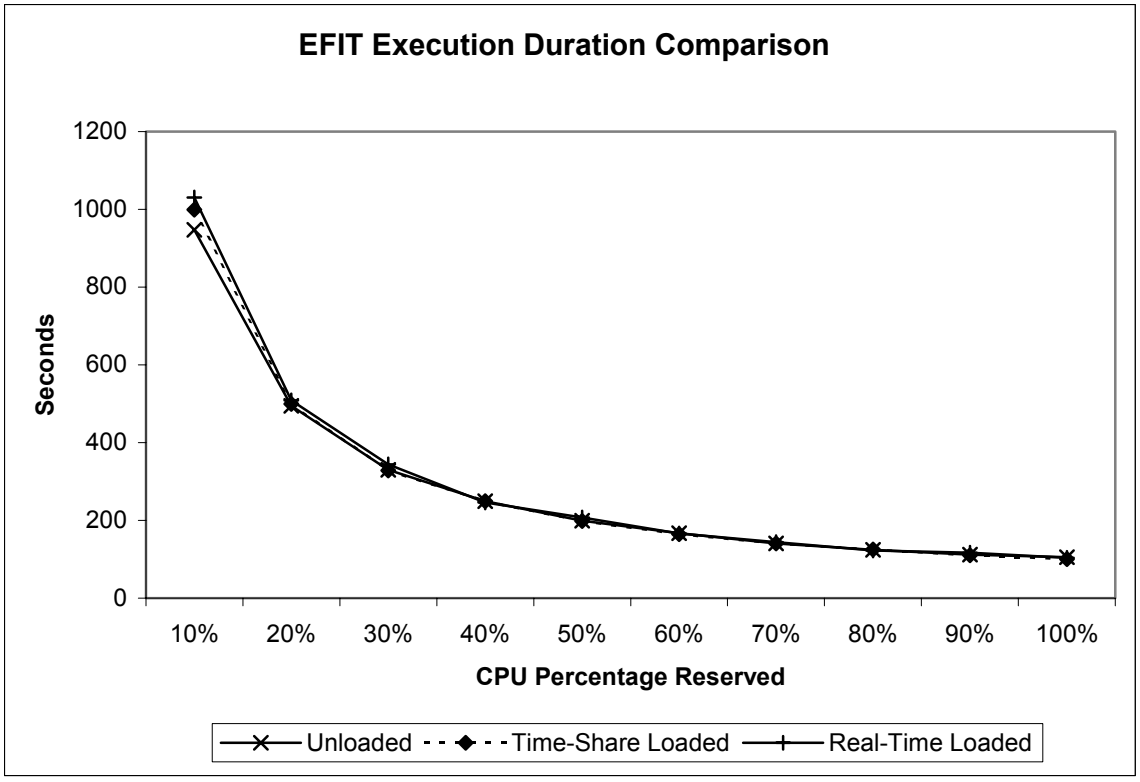

Figure 3: Results under varying load conditions

References

1. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of High Performance Computing Applications, 2001. **15**(3): p. 200-222.
2. Keahey, K., T. Fredian, Q. Peng, D.P. Schissel, M. Thompson, I. Foster, M. Greenwald, and D. McCune, *Computational Grids in Action: the National Fusion Collaboratory.* Future Generation Computing Systems (to appear), October 2002. **18**(8): p. 1005-1015.
3. Tao, L., *Shifting Paradigms with the Application Service Provider Model.* IEEE Computer. **34**(10): p. 32-39.
4. Tuecke, S., K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman, *Grid Services Specification (Draft 3, 7/17/2002)*. 2002, http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft03_2002-07-17.pdf.
5. I. Foster, C. Kesselman, J. Nick, and S. Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.* available at www.globus.org/ogsa, 2002.
6. *The National Fusion Collaboratory.* http://www.fusiongrid.org.
7. Lao, L.L., H. St. John, R.D. Stambaugh, A.G. Kellman, and W. Pfeiffer, *Reconstruction of Current Profile Parameters and Plasma Shapes in Tokamaks.* Nucl. Fusion, 1985. **25**: p. 1611.
8. J. Ongena, M.E., D. McCune, *Numerical Transport Codes.* Proceedings of the Third Carolus Magnus Summer School on Plasma Physics, March, 1998. **33**: p. 181-191.
9. Ludwig, H., A. Keller, A. Dan, and R.P. King, *A Service Level Agreement Language for Dynamic Electronic Services.* IBM Research Report RC22316 (W0201-112), January 24, 2002.
10. K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, *SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems.* 8th Workshop on Job Scheduling Strategies for Parallel Processing, July 2002.
11. Keahey, K. and V. Welch, *Fine-Grain Authorization for Resource Management in the Grid Environment.* to appear in the Proceedings of Grid2002 Workshop, November 2002.
12. Nahrstedt, K., H. Chu, and S. Narayan. *QoS-aware Resource Management for Distributed Multimedia Applications*. in *Journal on High-Speed Networking, IOS Press*. December 1998.
13. Foster, I., C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation*. In *Proc. International Workshop on Quality of Service*. 1999.

14. Valerie Taylor, X.W., Xin Li, Jonathan Geisler, Zhiling Lan, Mark Hereld, Ivan R. Judson and Rick Stevens, *Prophesy: Automating the Modeling Process.* Annual International Workshop on Active Middleware Services, August 2001.
15. Gannon, D., R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski, *Grid Web Services and Application Factories.* (notes available at http://www.extreme.indiana.edu/xgws/afw/), 2002.
16. Arnold, D.C. and J. Dongarra, *The NetSolve Environment: Progressing Towards the Seamless Grid.* Proceedings of the International Conference on Parallel Processing (ICPP-2000), 2000.
17. Satoshi Matsuoka, H.N., Mitsuhisa Sato, Satoshi Sekiguchi, *Design issues of Network Enabled Server Systems for the Grid.* GRID 2000, Springer-Verlag,: p. 4-17.
18. Weissman, J.B. and B.-D. Lee, *The Virutal Service Grid: An Architecture for Delivering High-End Network Services.* In Concurrency: Practice and Experience, 2002.
19. Loyall, J.P., R.E. Schantz, J.A. Zinky, and D.E. Bakken. *Specifying and Measuring Quality of Service in Distributed Object Systems*. in *ISORC*. 1998. Kyoto, Japan.
20. Raje, R.R., B.R. Bryant, A.M. Olson, M. Auguston, and C. Burr, *A Quality-of-Service-Based Framework for Creating Distributed Heterogeneous Software Components.* Concurrency and Computation: Practice and Experience, 2002. **14**: p. 1009-1034.
21. Gu, X. and K. Nahrstedt. *A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids*. in *11th IEEE International Symposium on High Performance Distributed Computing*. July 2002.