# Analysis and Provision of QoS for Distributed Grid Applications

Rashid J. Al-Ali[1], Kaizar Amin[2,3], Gregor von Laszewski[2], Omer F. Rana[1], David W. Walker[1], Mihael Hategan[2], and Nestor Zaluzec[2]
[1] *Cardiff University, UK*
[2] *Argonne National Laboratory, USA*
[3] *University of North Texas, USA*

**Abstract.**

Grid computing provides the infrastructure necessary to access and use distributed resources as part of virtual organizations. When used in this way, Grid computing makes it possible for users to participate in collaborative and distributed applications such as tele-immersion, visualization, and computational simulation. Some of these applications operate in a collaborative mode, requiring data to be stored and delivered in a timely manner. This class of applications must adhere to stringent real-time constraints and Quality-of-Service (QoS) requirements. A QoS management approach is therefore required to orchestrate and guarantee the timely interaction between such applications and services. We discuss the design and a prototype implementation of a QoS system, and demonstrate how we enable Grid applications to become QoS compliant. We validate this approach through a case study of an image processing task derived from a nanoscale structures application.

**Keywords:** Grid Computing, Quality of Service, Resource Management

## 1. Introduction

Many commercial and scientific applications require access to high-performance and high-end resources that are both expensive to own and maintain. Often there are also a limited number of such resources available. The development of Grid infrastructure (Laszewski and Wagstrom; Foster, Kesselman, Tuecke) allows these resources to be shared by service providers and users. However, effective use of such resources necessitate the development of mechanisms which take into account the particular constraints needed to satisfy particular application resource demands. Hence, some applications may be dependent, to a greater degree, on obtaining results within a particular time frame. Visualisation applications are an example – where the rendering of a graphical scene requires simulation results to be returned within a particular time period for the application to be usable. Consequently, considerable effort has gone into the development of resource scheduling algorithms and complex execution frameworks to satisfy these requirements. Although the Grid community has collectively made significant progress towards

support for such types of applications, only recently has consideration been given to a fundamental problem prevailing in service-oriented architectures: providing deterministic Quality-of-Service (QoS) assurances to service consumers. After all, providing non-trivial QoS is one of the primary goals of the Grid approach – as without such support distributed resources become unusable. Nevertheless, most Grid environments operate on a best-effort basis, sharing the Grid resources among users with equal priority. Degradation in performance and efficiency is a key need that must be addressed when a large number of requests are issued for the same set of shared Grid resources.

To address this problem, we concentrate on ways to increase the collective efficiency of scientists using resources collaboratively. One practical solution is to introduce QoS mechanisms that enable service providers to partition their services based on quality criteria such as priority, fairness, and economic gain. In other words, a QoS-aware Grid infrastructure can offer deterministic QoS assurances based on a particular criterion, rather than on a best-effort basis. In previous work we outline the G-QoSm architecture for managing and specifying QoS attributes associated with Grid services (Al-Ali, Rana et al.), and a technique for integrating this with the the Java CoG Core kit (Al-Ali, Amin et al.). Our approach is restricted to handle soft-real time applications – whereby we provide support for making resource reservations to accomplish a particular execution deadline, but do not provide any guarantees that the actual deadline is met. We can only provide guarantees that a single (or collection of) resource(s) will be available to execute an application – but not that the subsequent execution will complete within a certain time.

The paper is structured as follows. In Section 2 we provide an overview of QoS in networking and computational resource sharing. In Section 3 we outline the general requirements of a Grid QoS management system and give an overview of existing Grid QoS systems. In Section 4 we present G-QoSm and its major components. In Section 5 we discuss a typical high-performance Grid application and outline its QoS requirements. In Section 7 we discuss performance results based on executing applications with QoS support. At present we primarily provide support for management of computational resources – although some initial work has been undertaken on supporting network resources also (using the Bandwidth Broker (S. Sohail et al.)).

## 2. Quality-of-Service: Background and Terminology

Quality of service has been explored in various contexts (Oguz et al.; Bochmann et al.). Two types of QoS attributes can be distinguished: those based on the quantitative, and the qualitative characteristics of the Grid infrastructure. Qualitative characteristics refer to aspects such as service reliability and user satisfaction. Quantitative characteristics refer to aspects such as network latency, CPU performance, or storage capacity. For example, the following are quantitative parameters for network QoS: Delay (the time it takes a packet to travel from sender to receiver), Delay jitter (the variation in the delay of packets taking the same route), Throughput (the rate at which packets go through the network), Packet-loss rate (the rate at which packets are dropped, lost, or corrupted). Although qualititative characteristics are important, it is difficult to measure these objectively. Systems which are centered on the use of such measures utilise user feedback (Deora et al.) to compare these, and relate them to particular system components. Our focus is primarily on quantitative characteristics.

Similarly, compute QoS can be specified based on how the computational (CPU) resource is being used – i.e. as a shared or an exclusive-access resource (Roy et al.). When more than one user-level application shares a CPU, the application can specify that it requires a certain percentage access to the CPU over a particular time period. In exclusive-access systems, in which usually one user-level application has exclusive access to one or more CPUs, the application can specify the number of CPUs as a QoS parameter. In exclusive-access only one application will be allowed to use the CPU for 100% of the time, over a particular time period.

Storage QoS is related to access to devices such as primary and secondary disks or other devices such as tapes. In this context, QoS is characterised by bandwidth and storage capacity. Bandwidth is the rate of data transfer between the storage devices and the application program reading/writing data. Bandwidth is dependent on the speed of the bus connecting the application to the storage resource, and the number of such buses that can be used concurrently. The number and types of parallel I/O channels available between the processor and the storage media are significant parameters in specifying storage QoS. Capacity is the amount of storage space that the application can use for writing data.

It is necessary for applications to specify their QoS requirements as the characteristics of a single resource that is necessary to run their application (compute, storage and network), and the period over which the resource is required. Such a resource may, in practise, involve the

aggregation of a number of different network, compute and data resources to achieve the desired outcome. Resource reservation provides one mechanism to satisfy the QoS requirements posed by an application user, and involves giving the application user an assurance that the resource allocation will provide the desired level of QoS. The reservation process can be immediate or undertaken in advance, and the duration of the reservation can be definite (for a defined period of time) or indefinite (from a specified start time and till the completion of the application).

## 3. QoS in Grid Computing

A key Grid problem that many researchers have been investigating is resource management, specifying how Grid middleware can provide resource coordination for client applications transparently. One of the most successful middleware projects that provides such coordination is the Globus Alliance (The Globus Alliance). Recently, there has been a push to make greater use of Grid middleware in business applications – as the traditional focus has been towards computational science. This change in emphasis has also led to greater emphasis being placed on commercial technologies – such as Web Services – and currently service-oriented concepts play a key role in emerging Grid standards.

Generally, Grid applications submit their requirements to Grid resource management services that schedule jobs as resources become available. Each resource provider must support a resource manager or scheduler that can receive requests from external applications (i.e. applications that are being managed by individuals who do not own the resources). However, there are several applications that need to obtain results for their tasks within strict deadlines, hence they cannot wait for resources to become available. For these applications, it is often necessary to reserve Grid resources and services at a particular time (in advance or on-demand). In addition, other features are highly desirable, indeed required, if the Grid resource management service is to be able to handle complex scientific and business applications. We review these requirements in the next subsection and then briefly discuss how well current QoS systems meet these requirements.

### 3.1. Requirements

A Grid resource management system attempt to address the following requirements that relate to QoS issues.

**Advance Resource Reservation** The system should support mechanisms for advance, immediate, or 'on-demand' resource reservation. Advance reservation is particularly important when dealing with scarce resources, as is often the case with high-end resources made available on the Grid.

**Reservation Policy** The system should support a mechanism that allows Grid resource owners to enforce their policies governing when, how, and who can use their resource. This should be undertaken while decoupling reservation and policy entities, in order to improve reservation flexibility (Karsten et al.).

**Agreement Protocol** The system should assure the clients of their advance reservation status, and the resource quality they expect during the service session. Such assurance can be contained in an agreement protocol, such as Service Level Agreements (SLAs).

**Security** The system should prevent malicious users penetrating, or altering, data repositories that hold information about reservations, policies and agreement protocols. In addition to a secure channel between an application and the Grid resources it uses, a security infrastructure that provides support for authentication, authorisation and access control should be provided.

**Simplicity** The QoS enhancement should have a reasonably simple design that requires minimal changes to be made to existing computation, storage or network infrastructure (although, in practise, this tradeoff is hard to achieve).

**Scalability** The approach should be scalable to a large number of entities, since the Grid is a global-scale infrastructure. This is especially true as Grids are expected to be open and dynamic, with resources and users joining and leaving the Grid in a non-deterministic manner.

### 3.2. Current QoS Efforts

Quality of Service (QoS) has been extensively explored in distributed multimedia and networking communities (Bochmann et al.; Oguz et al.). In the multimedia community, QoS issues are geared to provide a client with an acceptable level of presentation quality when accessing a multimedia document. This level of quality includes supporting network QoS connecting the client to the server, and end server QoS comprising compute and memory performance, to process and dispatch multimedia

frames at specific rates. In this context, network QoS deals specifically with providing certain quality levels for network link characteristics between two points, with these characteristics expressed in terms of delay, jitter, packet loss rate and throughput (bandwidth). To manage these network parameters, either particular network elements are modified – essentially network routers or switches – to support specialist protocols (such as RSVP), or changes are made at the network end-points to control how packets from an application are transmitted based on feedback from the receiver. The first of these (based on modifying network elements) is often not a viable option, as network administrators are reluctant to make such modifications. The alternative approach is often the preferred solution, whereby feedback about network performance is used to control the rate at which data is transmitted at the sender end. Such a variable rate mechanism may lead to a reduction in the presentation quality of the delivered data, but allows at least some output to be generated at the receiver end (compared to no output at all).

In Grid computing, QoS management aims to provide assurance for accessing resources, while maintaining the security level between domains. Unlike multimedia and network QoS, Grid QoS requires a central information service (Czajkowski et al.) for up-to-date information on resources available for use by others. Such an information service can be interrogated by an application user to determine which resources can be used to execute an application. As Grid QoS simultaneously deals with a number of resources per service session, Service Level Agreements (SLAs) become essential to specify the service level the client must receive and the provider must supply. Such SLAs must also make use of parameters that are provided by resource owners when they publish properties of their resources. It is important that each parameter within the SLA is capable of being monitored. SLAs often encode requirements that an application user wishes to achieve, and capabilities that a resource owner can provide to others. Such contracts between users and providers may be expressed using first-order logic, algebraic operators, or be encoded within a scripting language as a policy. Often there is a tradeoff between the expressiveness offered by a particular encoding style, and the ease of use, evaluation and modification of a particular requirement. QoS management in Grid computing has recently become an active area of research.

Sahai et al. (Sahai et al.) propose a SLA management entity to support QoS in the context of commercial Grids. They envision the SLA management entity existing within the OGSA architecture, with its own set of protocols for manageability and assurance; they also describe a language for SLA specification. Although an interesting ap-

proach, this work is still at a very preliminary stage, and the general applicability of this work is still not obvious.

A general negotiation model called Service Negotiation and Acquisition Protocol (SNAP) is introduced be Czajkowski et al. (Czajkowski et al.), which proposes a resource management model for negotiating resources in distributed systems such as Grids. SNAP defines three types of SLAs that coordinate management across a desired resource set, and can, together, be used to describe a complex service requirement in a distributed environment. Resource interactions are mapped to well-defined, platform-independent, Service Level Agreements (SLAs), with the SNAP protocol managing resources across different administrative domains, via three types of SLAs: Task SLA (TSLA), Resource SLA (RSLA) and Bind SLA (BSLA). The TSLA describes the task that needs to be executed, and the RSLA describes the resources needed to accomplish this task. The BSLA provides an association between the resources from the RSLA and the application 'task' in the TSLA. The SNAP protocol necessitates the existence of a resource management entity that can provide guarantees on resource capability; for example, RSLA. Our reservation model can encapsulate such a requirement and implement the RSLA negotiation. TSLA is similar to our QoS request, where the user provides the resource requirements along with the service desired.

Keahey et al. (Keahey et al.) propose an architecture called Virtual Application Service (VAS) for managing QoS in computational Grids. VAS is an extended Grid service with additional interfaces for negotiation of QoS level and service demands. The key objective of VAS is to facilitate the execution of real-time services which have very specific deadline constraints. A client submits a request to VAS for advance or immediate reservation of a service; supplying only time constraints. The system has metadata – consisting of application information and application modelling information associated with every service, allowing the system to compute the feasibly of fulfilling the client's request under such time constraints. From this metadata (such as execution time and hardware resource information) the system determines the computational (CPU) resources required to support the request, and, subsequently, undertakes CPU slot reservation. A Service Level Agreement is then presented to the user. This work is similar to our approach with the following differences:

- VAS is a deadline-bound system, and the client must only specify the time constraints as the QoS metric.
- VAS is designed for a specific application domain called National Fusion Collaboratory (NFC).

- VAS assumes that with every service deployed, there must be metadata, and some application modelling (such as execution time with specific hardware) has been previously undertaken. It therefore requires the application to predict how long it will need to run. This is a particualarly difficult objective to achieve in a dynamic Grid environment.
- VAS computes the time needed for service execution, based on a prediction model and uses service metadata as a basis for this.

In our approach we make use of different allocation strategies for running user applications – based on whether they require a *Time-domain* or *Resource-domain* allocation strategy. For users who request a Time-domain allocation, 100% of the computational resources must be allocated to their jobs. Therefore, whereas VAS requires users to benchmark their applications by running them first on an unloaded (100%) CPU – to enable prediction of their execution times when the CPU also contains other application – our approach does not necessitate this benchmarking. Instead, we can utilise results of application execution times where a guaranteed service execution has been requested – and use these as a benchmark. Although the difference is subtle, but is nevertheless important in a dynamic and open environment, such as the Grid, where the need to execute code to benchmark their times cannot be enforced on application users. Burchard et al. (Burchard et al.) also propose the use of SLAs to negotiate service execution parameters between resource managers. The SLA management is achived via a Virtual Resource Manager (VRM) – that enables interaction between a number of schedulers on different clusters. The VRM acts as a coordinator to aggregate SLAs negotiated with different sub-systems. Although the SLA management in this work is similar to our effort, the focus in our approach is on utilising the service paradigm, where the VRM is intended to integrate execution across a number of co-located clusters.

Members of the the Global Grid Forum (GGF) have started to identify issues of concern related to Grid QoS. For example; the GGF Grid Resource Agreement and Allocation Protocol (GRAAP) Working Group (WG), has produced a 'state of the art' document which lays down properties for resource reservation in Grids (MacLaren). We envision that our reservation model, employed in the proposed system, can be used to support the reservation properties outlined by the GRAAP-WG. The GRAAP-WG has already produced a GGF draft recommendation document, and still under discussion is an OGSI-based agreement model (Czajkowski et al.). This proposed Agreement-based Grid Service Management (OGSI-Agreement) model, defines a set of

OGSI-compatible `portTypes` through which management applications and services can negotiate. This agreement defines the required behavior (QoS) of a delivered service, with reference to a particular service consumer. Further, the document contains an abstract management protocol to manage the agreement stages of the QoS lifecycle – from service creation up until termination. Once the agreement model has been standardized by the GGF, we will attempt to incorporate this agreement model into our G-QoSm framework. Although it is not exactly clear how much of this work will actually come to fruition, with the current emphasis on modifying OGSI to be compliant with Web Services (as part of the recent Web Services Resource Framework (WSRF) effort).

The General-purpose Architecture for Reservation and Allocation (GARA) is the most commonly known framework for supporting QoS in the context of computational Grids. GARA (Foster, Kesselman, Lee et al.) provides programmers with the capability to specify end-to-end QoS requirements. It provides advance reservations, with uniform treatment of various types of resources such as network, computation, and storage. GARA's reservation is aimed to provide a guarantee that the client or application initiating the reservation will receive a specific QoS from the resource manager. GARA also provides an application programming interface to manipulate reservation requests, such as *create*, *modify*, *bind*, and *cancel*. GARA uses the Dynamic Soft Real-time (DSRT) scheduler (Chu et al.) as the underlying resource manager for computational resources.

Although GARA has gained popularity in the Grid community, it has limitations in coping with current application requirements and technologies:

- The current focus in Grid computing is towards the use of Web Service technologies. The road map of many existing Grid middleware systems also suggests a move towards Web services standards – such as the increasing importance of the Web Services Resource Framework (WSRF). GARA is not OGSA-compliant, and therefore, OGSA-enabled applications cannot directly make use of GARA services.

- Grid applications require the simultaneous allocation of various resources. An agreement protocol should exist to inform the application about the resources negotiated for allocation and the level of quality the application expects. This information is usually encapsulated in a Service Level Agreement (SLA). GARA does not support the concept of an agreement protocol and establishing a SLA for various resources.

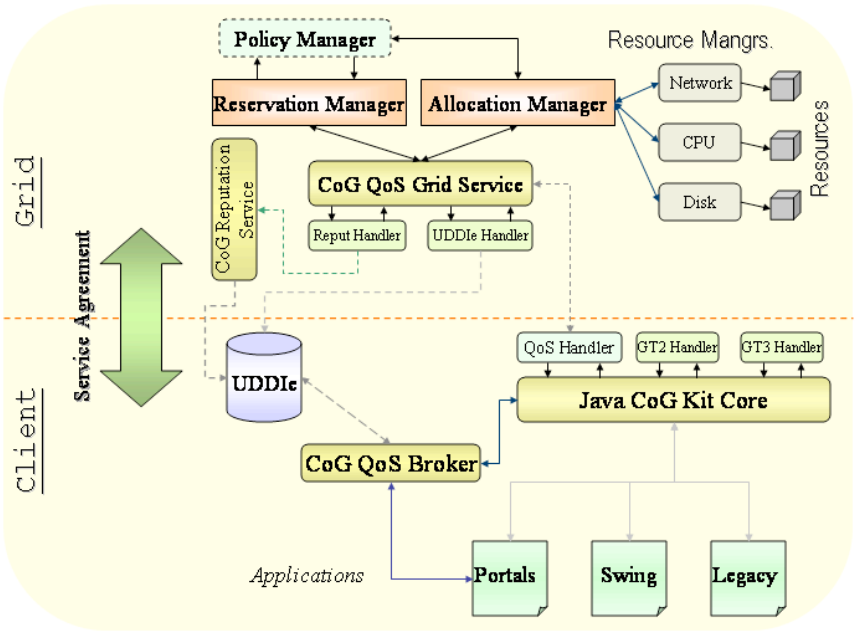


*Figure 1.* The G-QoSm architecture with an OGSA-enabled QoS service.

– QoS monitoring and adaptation during the active QoS session is one of the most important and successful mechanisms to provide a quality guarantee (Al-Ali, Hafid et , Rana et al.). GARA is not tooled with adaptive functions to support compute resources.

## 4. Grid QoS Management

Grid Quality of Service Management (G-QoSm) is a framework to support QoS management in computational Grids in the context of the Open Grid Service Architecture (OGSA) (Al-Ali, Rana et al.; Al-Ali, Amin et al.). G-QoSm consists of three main operational phases: establishment, activity, and termination. During the *establishment* phase, a client application states the desired service and QoS requirements. G-QoSm then undertakes a service discovery, based on the specified QoS properties, and negotiates an agreement for the client application. During the *activity* phase additional operations such as QoS monitoring, adaptation, accounting and possibly re-negotiation may take place. During the *termination* phase the QoS session is ended due to resource reservation expiration, agreement violation, or service completion; resources are then freed for use by other clients. The framework supports

these three phases using a number of specialist components, as depicted in Figure 1. In subsequent sections we describe these interactions, and highlight how service provision is undertaken.

### 4.1. QoS Grid Service

The basic building block of our architecture is the QoS Grid service (QGS), an OGSA-compliant Grid service providing QoS functionalities such as negotiation, reservation and resource allocation with certain quality levels. Each QoS-enabled resource is accessed through a QGS. The QGS publishes itself to a QoS registry service, so clients and QoS brokers are able to discover the existence of the QGS. In addition to the QoS functionalities, it supports two types of allocation strategies:

- Resource domain: in this allocation strategy a client can specify a certain percentage capacity of the shared QoS-enabled resource, e.g. access to 50% CPU time, or a 20Mbps bandwidth out of a total of 155Mbps available.
- Time domain: in this allocation strategy a client may request an entire resource to be reserved for exclusive use, i.e. no other clients/applications are allowed to share the resource; for example; reserving 100% of the CPU.

This functionality is enabled by ensuring that all requests for resources are issued through the QGS. Further, the QGS interacts with a number of modules to deliver QoS guarantees. These modules are the QoS Handler, reservation manager, allocation manager, and the QoS registry service (see Figure 1). Currently, G-QoSm supports compute resource based QoS only (we have started to also include reservation of network-based resources via a Bandwidth Broker, and will subsequently attempt to support disk allocation).

The architecture, as illustrated in figure 1 consists of a client (bottom part of the figure) and a service provider (top part of the figure). The client makes use of a registry service (UDDIe), and may be implemented using the Java CoG Kit. The client may be a physical user accessing the registry service via a portal, or may be another service issuing a search request. If the client is another service, access to the registry is via a *QoS Broker*. A service provider, on the other hand, illustrated in the top portion of figure 1 must provide access to physical resources that are used to manage the service (this includes support for computation, data storage and network access). The first interaction between a client and a service provider is therefore via the discovery operation invoked on the registry service. The UDDIe Handler enables

a service provider to publish properties of a service within the registry, and to subsequently alter any parameters associated with the service. Once a request for a service has been received, the reservation or allocation manager is invoked. To support QoS characteristics, therefore, a service provider must ensure that in addition to the service being offered to external users, it also support additional components to allow reservation (and subsequently allocation) of resources on which the service is to be hosted. In addition, the service must be annotated with additional properties that enables these QoS attributes to be encoded in its interface.

The QGS performs: i) resource reservation and ii) resource allocation. When a reservation request is received, the QGS undertakes an admission control to check the feasibility of granting such a request. This feasibility check is undertaken by the reservation manger. If such a reservation is possible, the requested resources are reserved, the reservation table is updated and an agreement comprising the reservation specification is generated and returned to the client.

One the other hand, when a resource allocation request is received, the QGS verifies that the user has indeed made a reservation based on the supplied agreement. If this test passes, then the QGS submits the specification of the job to be executed to the Globus Resource Allocation Manager (GRAM) on that particular resource. Along with the job specification, the QGS supplies other parameters related to compute resource allocation with quality levels – these parameters are passed from GRAM to the compute resource manager for immediate allocation. This process is handled by the Allocation manager provided within the QGS. The QGS therefore contains the following modules:

– *Reservation Manager:* the reservation manager uses a data structure that supports reservations for quantifiable resources – i.e. resources associated with defined capacities. The reservation manager is de-coupled from the underlying resources and does not have direct interaction with them. However, it obtains resource characteristics, and policies governing resource usage, from the policy manager. The policy manager, on the other hand, is responsible for validating reservation requests by applying domain-specific rules, established by the resource owners, on when, how, and by whom the resource can be used. In brief, when the reservation manager receives a reservation request from the QGS, it contacts the policy manager for validation and then performs admission control to check the availability of the requested resource. Upon success, it returns a positive reply to the QGS, which allows the QGS to propose a negotiable service agreement offer.

– *Allocation Manager:* The Allocation Manager has a primary role to interact with underlying resource managers for resource allocation and de-allocation, and to inquire about the status of the resources. It has interfaces with various resource managers, namely, the Dynamic Soft Real Time Scheduler (DSRT) (Chu et al.) and Network Resource Manager (NRM); we are also investigating *Nest* as the disk storage resource manager (Bent et al.). When the allocation manager receives resource allocation request from the QGS, it forwards the request to the designated underlying resource manager. The Allocation Manager interacts with adaptive services to enforce adaptation strategies; for details, see (Al-Ali, Hafid et , Rana et al.). The NRM is implemented using the DiffServ Bandwidth Broker (S. Sohail et al.)

– *QoS Registry Service:* Since the framework is based on the OGSI implementation, the QGS and other Grid services in the OGSI container should be published in some registry service so they can be accessed by others. Service publishing, in this discussion, does not mean simply publishing a service name, URL, and basic description. For example, for QGS, it includes information on what QoS-enabled service it offers, what allocation strategies it employs, and what classes of network QoS it offers (e.g., best effort, controlled load, or guaranteed). For other Grid services, service publishing includes information about QoS properties such as performance characteristics and service execution requirements. We make use of an extended version of the Universal Description Discovery and Integration (UDDI) registry to achieve this. UDDIe (ShaikAli et al.) is a Web services registry which provides service providers a means to publish their services with QoS properties and, hence, to search for these services based on the QoS properties.

### 4.2. Java CoG Kit Core

The QoS Grid Service (QGS) described offers the requisite functionality for QoS-related features that allows the provisioning of any arbitrary Grid resource into a QoS-aware Grid entity. However, in order to take advantage of such QoS-aware Grid resources it is important for applications to conveniently interact with such entities without having to undergo significant changes in logic and implementation. Hence, we provide interactions with the QGS via convenient middleware libraries making it seamless for Grid applications to benefit from the G-QoSm architecture.

The Java CoG Kit (Laszewski and Foster et al.) is a Java-based modular middleware used to provide access to various Grid implementations such as Globus Toolkit v2 (GT2) and v3 (GT3). One of the modules of the Java CoG Kit, called as *cog-core*[1], provides the core functionality for such technology- and architecture-independent interoperability. Cog-core provides APIs offering abstract Grid functionality such as remote job execution and file transfers without any consideration for the underlying Grid implementation. For example, consider a Grid application developed using the APIs provided by cog-core. Since cog-core offers absolutely abstract functionality irrespective of the backend architecture, the same application can be executed on a variety of platforms. Hence, to run the application on a GT2 service, the user needs to merely mention a provider attribute as GT2. The same application can be later executed on a GT3 service without any modification to its implementation by simply changing the provider attribute from GT2 to GT3. Cog-core contains the required functionality to map the abstract application requirements into backend specific details controlled by the corresponding provider attribute.

In order to provide seamless interaction between Grid applications and the QoS-aware Grid resources, we have augmented the functionality of cog-core to incorporate QoS-related parameters. Thus, all the necessary logic and implementation overhead for QoS management is embedded into the cog-core, thereby allowing the applications to enjoy QoS features by simply changing the provider attribute to "QoS". In the rest of this section we describe the basic [2] constructs of the cog-core library and its enhancement into the QoS domain.

– *Task:* Cog-core defines a *Task* as an atomic unit of execution. It abstracts the generic Grid functionality including authentication, remote job execution, file transfer request, and information query. It has a unique identity, a security context, a specification, a service contact, and a provider attribute.

  The task identity helps in uniquely representing the task across the Grid. The security context represents the abstract security credentials of the task. Apparently, every backend Grid implementation will have its own notion of a security context. Hence, the security context in cog-core offers a common construct that can be extended by the different implementations to satisfy the corresponding back-end requirement. The specification represents the actual attributes or parameters required for the execution of the Grid-centric task. The generalized specification can be extended

[1] Formerly known as the GridSDK (Amin et al.)
[2] for a detailed understanding of cog-core the reader is directed to (Amin et al.)

for common Grid tasks such as remote job execution, file transfer, and information query. The service contact associated with a task symbolizes the Grid resource (service) required to execute it. As mentioned earlier, the provider attribute specifies the desired backend Grid implementation for the Task.

– *Handlers:* The *Task Handler* provides a simple interface to handle interaction with a generic Grid task. It categorizes the tasks and provides the appropriate functionality for it based on the provider attribute of the submitted task. Cog-core contains a separate handler for every backend functionality it supports. These handlers then map the generic Grid parameters of the Task into the backend implementation specific Grid functionality.

In order to augment the cog-core functionality into the QoS domain we provide a QoS handler that encapsulates the QoS-related implementation and logic. The QoS task handler manages the QoS negotiation, re-negotiation, task execution, and data redirection between the end application and the QoS-aware Grid resource (QGS).

### 4.3. Negotiation of QoS Levels

The QoS negotiation process aims to reach some agreement between the client and the service provider on either the reservation schedule, or the parameters involved in providing a given service. It is not necessary for such negotiation to take place everytime – especially if the service provider can meet the request made by a client immediately. However, if the constraints identified in the service request by a client cannot be met (such as the desired time of service, the active period of the service and resource characteristics needed for the service, etc – the 'QoS levels'), it is necessary for the service provider and client to agree on some mutually agreeable constraints. It is these levels that are negotiated during this process.

The QoS negotiation is, therefore, essentially a *matchmaking* process between the client's desired QoS constraints, and the service provider's resource capacity, to ensure the request does not exceed such resource capacity. A client may request constant QoS levels during the lifetime of a service session. For example, a data transfer service is negotiated to transfer a data set from point $A$ to $B$ at *100 Mbps* – but during the transfer session, it may be possible that the requested bandwidth cannot be sustained. The client may request a decrease of the requested bandwidth while the transfer service is active – or the service provider must find additional capacity to sustain the QoS demands of the client if an agreement has been reached. It is therefore possible for a client

to realise that the initial 100Mbps cannot be achieved, and request for a lower bandwidth – leading to a re-negotiation. If the client's re-negotiation request has lower QoS levels than the original request, then the new request is guaranteed, but if the re-negotiation request increases the QoS level, the service provider has to run an admission control check, treating the request as a new QoS negotiation, subject to approval or rejection (based on the other services it is managing at the time).

The QoS negotiation process involves two aspects: 1) *service negotiation* and 2) *QoS negotiation*. Decoupling service and QoS negotiations improves system availability and flexibility; as system availability is concerned with the number of requests admitted, while system flexibility is concerned with adapting to different client requests during an active QoS session. Further discussion on QoS adaptation can be found in our previous work (Al-Ali, Hafid et , Rana et al.).

The proposed QoS negotiation model necessitates that the client undertake a service negotiation phase, with the QoS negotiation phase being optional for negotiating resource characteristics and quality levels. Two mechanisms are envisaged to obtain resource characteristics and the required qualities for the negotiated service, where the client:

- explicitly supplies resource characteristics and qualities required, or
- relies on the service profile stored in the QoS registry (ShaikAli et al.).

In the latter case, the system generates the service profile from either the service provider, a third-party reputation service, statistics based on the feedback provided by a client, or uses prediction models, (such as (Jarvis et al.)). Quality levels within the service profile are dynamically updated and stored in the QoS registry. The service profile is for use by the QoS manager where a client either specifically requests services with the default profile, or does not have details on the resource quality required. In such cases, a service profile contains a set of quality levels suggested for the negotiated service.

#### 4.3.1. *QoS Negotiation Protocol*

The negotiation protocol specifies the syntax and semantics of the message exchange between the entities involved in the negotiation process, aimed at reaching mutual agreement between the entities involved. These entities include the client, the QoS service and the service provider. It is important to note that the QoS service is the central entity with the role of coordinating the negotiation process between

client and provider. It is further assumed that the provider delegates the QoS service to act on its behalf, and, therefore, there is no direct interaction between the client and the provider during negotiation.

The QoS service supports a number of operations for use by the client, with the interaction between the client and QoS service based on XML message exchange. In this discussion the operations related to the negotiation process are presented in the sequence *Query*, *Reserve*, *Update* and *Cancel*.

- Query: the QoS service maintains information about resources available for use by clients (in a registry service). The *Query* operation is used to interrogate the registry to find a service that has particular QoS attributes. If a suitable service is found, the QoS service will hold the resource(s) for a limited period (a temporary reservation) and returns a *query handle*. The resource(s) are held until the client confirms the reservation or the temporary reservation time elapses. If the query operation cannot find the required service (i.e. all QoS attributes in the query do not match), it tries to find a service that matches on *most* of the attributes. There may be one or more services which match this description. The QoS service now returns a new service offer as a counter-proposal. Figure 2 is an XML schema definition for the *Query* operation syntax.

- Reserve: after a successful *Query* operation, and, while the resources are held on a temporary basis, the *Reserve* operation is used to confirm the queried resources. Subsequently, the QoS service will reserve the resources for the specified period and returns an *agreement handle* to be used during service invocation. Figure 3 is an XML schema definition for the *Reserve* operation syntax.

- Update: the *Update* operation is used for re-negotiation purposes. If a client wishes to modify the constraints on particular QoS attributes (during an active session), then the client would use this operation. When relaxing QoS constraints the operation is guaranteed, however if QoS constraints are modified in such a way that additional resources are required, the re-negotiation request is treated as a new request (meaning an admission control procedure will be applied to ensure the requested resources do not exceed the capacity), and the request is, therefore, subject to approval or rejection – equivalent to a *Query* operation followed by a *Reserve* operation. Figure 4 is an XML schema definition for the *Update* operation syntax.

```
<xs:element name="Query">
        <xs:annotation>
           <xs:documentation>XML Schema for Query Operation</xs:documentation>
        </xs:annotation>
        <xs:complexType>
           <xs:sequence>
              <xs:element name="service">
                 <xs:complexType>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="type" type="xs:string" use="optional"/>
              </xs:complexType>
               </xs:element>
               <xs:element name="temporalQoS">
                  <xs:complexType>
                 <xs:attribute name="startTime" type="xs:dateTime" use="required"/>
                 <xs:attribute name="endTime" type="xs:dateTime" use="required"/>
              </xs:complexType>
               </xs:element>
               <xs:element name="computeQoS" minOccurs="0">
                  <xs:complexType>
                 <xs:attribute name="capacity" type="xs:integer" use="required"/>
                 <xs:attribute name="nodeCount" type="xs:integer" use="optional"/>
                  </xs:complexType>
              </xs:element>
              <xs:element name="networkQoS" minOccurs="0">
                   <xs:complexType>
                 <xs:attribute name="sourceIP" type="xs:string" use="required"/>
                 <xs:attribute name="destIP" type="xs:string" use="required"/>
                 <xs:attribute name="bandwidth" type="xs:integer" use="required"/>
              </xs:complexType>
               </xs:element>
           </xs:sequence>
        </xs:complexType>
</xs:element>
```

*Figure 2.* XML Schema for Query Operation

– Cancel: the *Cancel* operation cancels an *agreement handle* made by a *Reserve* operation, i.e. cancel reservation, which may only be used before the service session starts. If the service session has started, there is a different operation, not part of the negotiation process, which may be used to release resources. Figure 5 is an XML schema definition for the *Cancel* operation syntax.

The four operations of *Query*, *Reserve*, *Update* and *Cancel* are the fundamental elements of the negotiation model and define the protocol for the negotiation. When the QoS service receives a *Query* operation it performs an admission control procedure, to check whether the assigned resources plus the requested resources do not exceed the provider's maximum resource capacity. If the admission control passes, it returns a *query handle*, or reference, to be used for a subsequent *Reserve* operation. The *Reserve* operation does the actual resource reservation and returns an *agreement handle*. The client may cancel a previously-made *agreement handle*, for a session not yet started, by a *Cancel* operation. The client can re-negotiate a QoS session through an *Update* operation. These operations are suitable and sufficient for a flexible negotiation process in a distributed system model. Figure 6 is a sequence diagram for the QoS negotiation protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="Reserve">
      <xs:annotation>
        <xs:documentation>XML Schema for Reserve Operation</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:element name="serviceOffer">
            <xs:complexType>
              <xs:attribute name="queryHandle" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

*Figure 3.* XML Schema for Reserve Operation

### 4.4. Application Integration

In this section we walk through a sample scenario for an application to execute a QoS-enabled remote job. The application developer needs to specify the QoS parameters that must be considered during QoS negotiation. These parameters include start time, end time, resource type, and specifications. Once the Task object has been specified, the QoS Handler is delegated on behalf of the client or application to negotiate QoS requests. In this case the QoS Handler is seen as the client from the QGS point of view. This is a useful approach especially when the application requires more than one Grid resource. All that the application needs is to instantiate the required number of QoS Handler objects, submit the Task object to the handlers, and let the handlers negotiate QoS requests with the QGS to return an agreement.

Once the QoS parameters are successfully negotiated, the application then formulates the actual Grid Task that needs to be executed and submits it to the QoS handler along with the negotiation token (agreement). Furthermore, for QoS-enabled job submission through the interactive mode, the QoS handler listens for notifications of job status, with the notification implemented by the QGS as an OGSA notification.

```
<xs:element name="Update">
        <xs:annotation>
            <xs:documentation>XML Schema for Update Operation</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="agreement">
                    <xs:complexType>
                        <xs:attribute name="agreementHandle" type="xs:string" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="newTemporalQoS" minOccurs="0">
                    <xs:complexType>
                        <xs:attribute name="endTime" type="xs:dateTime" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="newComputeQoS" minOccurs="0">
                    <xs:complexType>
                        <xs:attribute name="capacity" type="xs:integer" use="required"/>
                        <xs:attribute name="nodeCount" type="xs:integer" use="optional"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="newNetworkQoS" minOccurs="0">
                    <xs:complexType>
                        <xs:attribute name="bandwidth" type="xs:integer" use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

*Figure 4.* XML Schema for Update Operation

The following code snippet shows a Java code fragment demonstrating how an application can generate a QoS negotiation request and QoS job submission to a QoS handler respectively.

```
 /*** QoS: Prepare Negotiation Task ***/
private void prepareQosNegotiationTask() {
  // create a QoS service, and setup QoS attributes
  Task task =
     new QosTaskImpl(‘‘myTask’’, QoS.NEGOTIATION);
  this.task.setAttribute(‘‘startTime’’, startTime);
  this.task.setAttribute(‘‘endTime’’, endTime);
  this.task.setAttribute(‘‘allocStrategy’’,strategy);
  this.task.setAttribute(‘‘cpu_capacity’’, cpuCapacity);

  // create a Globus version of the security context
  SecurityContextImpl securityContext =
      new GlobusSecurityContextImpl();
  // selects the default credentials
  securityContext.setCredential(null);
  // associate the security context with the task
  task.setSecurityContext(securityContext);
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
           attributeFormDefault="unqualified">
   <xs:element name="Cancel">
      <xs:annotation>
         <xs:documentation>XML Schema for Cancel Operation</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence>
            <xs:element name="agreement">
               <xs:complexType>
                  <xs:attribute name="agreementHandle" type="xs:string" use="required"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

*Figure 5.* XML Schema for Cancel Operation

```
  // create a contact for the Grid resource
  Contact contact = new Contact(‘‘myGridNode’’);

  // create a service contact
  ServiceContact service =
      new ServiceContactImpl(qosServiceURL);
  // associate the service contact with the contact
  contact.setServiceContact(‘‘QGSurl’’,service);

  // associate the contact with the task
  task.setContact(contact);
}

/*** QoS: Prepare Job Submission Task ***/
private void prepareQosJobSubmissionTask() {
  // create a QoS JobSumbission Task
  Task task =
     new TaskImpl(‘‘myTask’’, QoS.JOBSUBMISSION);
  this.task.setAttribute(‘‘agreementToken’’, token);
```

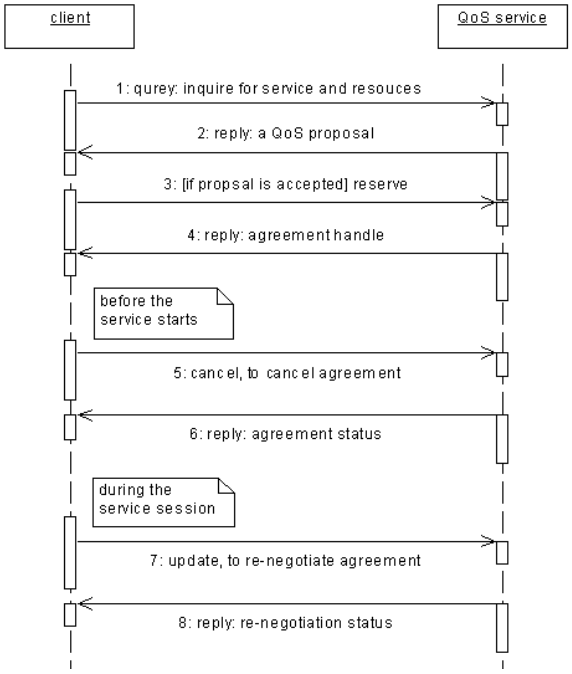


*Figure 6.* Sequence Diagram for the QoS Negotiation Protocol

```
// create a remote job specification
JobSpecification spec = new JobSpecificationImpl();

// set all the job related parameters
spec.setExecutable(‘‘/bin/myExecutable’’);
spec.setRedirected(false);
spec.setStdOutput(‘‘QosOutput’’);

//associate the specification with the task
task.setSpecification(spec);

// create a Globus version of the security context
SecurityContextImpl securityContext =
          new GlobusSecurityContextImpl();
 securityContext.setCredential(null);
task.setSecurityContext(securityContext);

Contact contact = new Contact(‘‘myQoScontact’’);

ServiceContact service =
      new ServiceContactImpl(qosServiceURL);
```

```
  contact.setServiceContact(‘‘QGSurl’’,service);
  task.setContact(contact);
}

/*** QoS: Task Submission to QoS Handler ***/ private void
QosTaskSubmission(Task task) {
  TaskHandler handler = new QoSTaskHandlerImpl();
  // submit the task to the handler
  handler.submit(task);
}
```

Abstracting the QoS services and interacting with the QGS by creating a task (i.e., QoS function) and submitting it to a QoS handler is of benefit when dealing with multiple distributed Grid resources. This approach makes the design and specification of abstract QoS-based brokers easier.

## 5. Application Case Study: Nanoscale Structures

To validate our QoS approach, we used our reference implementation of G-QoSm to manage a nanoscale structures application. This application involves a new experimental technique, position-resolved diffraction, being developed as part of Argonne National Laboratory's advanced analytical electron microscope program (Zaluzec). With this technique, a focused electron probe is sequentially scanned across a two-dimensional field of view of a thin specimen. At each point on the specimen a two-dimensional electron diffraction pattern is acquired and stored.

Analysis of the spatial variation in the electron diffraction pattern of each measured point allows the researcher to study subtle changes resulting from microstructural differences, such as ferro- and electromagnetic domain formation and motion, at unprecedented spatial scales. As much as one terabyte of data can be taken during such an experiment. The analysis of this data requires a resource-rich Grid infrastructure to accommodate real-time constraints. Results need to be archived, remote compute resources need to be reserved and made available during an experiment, and the data needs to be moved to the compute resources where they will be analyzed. Moreover, results need to be gathered and presented in a form that is meaningful to the scientist.

The need for a flexible infrastructure is demonstrated through a simple flow diagram depicted in Figure 7. The elementary logic of the instrument control can be expressed as a sequence of processes that

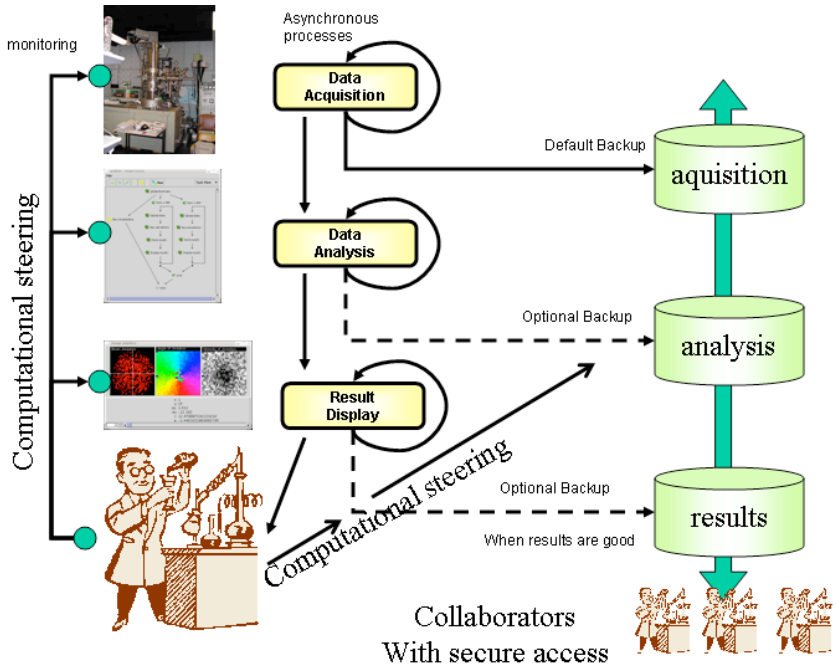


*Figure 7.* Asynchronous processes define a workflow steered by the scientist to support the problem-solving process with the help of abstract Grid tasks.

depend on each other: (a) Data acquisition: gathers time-delayed images from the electron microscope, (b) Backup: backs up the incoming data, (c) Data analysis: performs analysis on the time delayed images, and (d) Result display: gathers the results from the data analysis, in a form easy to interpret, to enable further judgments for steering the experiment.

The nanostructures application presents one of many scientific use patterns that occur in high-end instrument scenarios. The pattern includes a high volume of interaction during an experiment that must be dealt with in an adaptive and flexible way. Unexpected and unpredicted experiment conditions must be considered, and the instrument operator's interface to the Grid must be as simple as possible while at the same time provide the needed flexibility to interactively modify the experiment setup.

The Java CoG-Core Kit provides a convenient abstraction for formulating these tasks while reusing the patterns for file transfer, job execution, and job management. At the same time it hides much of the complexity, which the Grid application developer may not want to see. Graphical components for the Java CoG Kit are currently under development to achieve this, and it is expected that these will be integrated

via a problem-solving environment that targets the use of a scientific instrument. Through this interface, the scientist will be able to interact easily with the experiment resources and decide when, what, and where data gathered during the course of the experiment is backed up. Image filters and monitors, plugged dynamically into the workflow for image analysis, help to validate the correctness and usefulness of the running experiment. Since the sample in the instrument may require specialized and individual filters, the experiment operator must be given a methodology that allows their easy creation and adaptation. Because of the focus on the experiment itself, the use of the Grid should be through abstractions as much as possible. Based on the application description, we derive the following requirements for QoS: (a) network requirements to transfer the time-delayed images from the electron microscope as part of the data acquisition process, (b) disk storage to cache quickly incoming data during the acquisition process and the availability of large storage for a backup process, (c) computation power to process the scientific calculations on the time-delayed images in real time, as new images become available in the data analysis process, and (d) collect results produced by the data analysis process and transfer them to a display, where the scientist can interpret outcomes and further steer the experiment. Experiments undertaken on the QoS attributes are primarily targetted for category (c) and (d) in Section 7.

## 6. Case Scenarios and Requirements

In this section, we use case scenarios to illustrate how the QoSm framework can also benefit scientists in other disciplines.

- Collaborative Real-Time Experiments: A group of scientists located in different domains are collaborating on a nanostructures experiment. Each scientist participates in the experiment by providing local data *augmentation* and then transferring that data to a high-performance computing resource for collaborative data analysis. The scientists at corresponding domains establish a guaranteed network bandwidth to conduct data transfer; similarly, the scientists at the data analysis location establish resource guarantees, not only for the data transfer but also for computing power with adequate resources to perform the data analysis and produce results in a specific time, when all scientist are online to interact with or steer the experiment.
- Ad Hoc Real-Time Experiments Needing Computing Power: Several scientists decide to conduct an experiment to verify certain

findings. The decision is made on an ad hoc basis, that is, without prior arrangement. The experiment must be conducted in a Grid infrastructure, with enough computing resources to perform the desired experiment in a reasonable time and fulfill the scientists ad hoc requirements. Here, the scientists require some commitment from the Grid middleware that the resources needed for the experiment are indeed available at this time. The scientists therefore submit a QoS negotiation request to a QoS manager. The QoS manager gives such a commitment if the resources are available at the specific time; if the resources are not available, the QoS manager proposes a new available time, which the scientist may accept or reject.

– Experiments with Deadline Constraints: A team of scientists has a deadline for delivering experiment results. The scientists therefore contact the QoS manager in advance to negotiate a QoS agreement to guarantee resource availability during the experiment.

These three scenarios have the following common elements: (a) The need for Grid resources with particular capabilities (b) The need for resources to be available for a predefined period of time (c) The need for an agreement to indicate the commitment level of resource availability

With these elements in mind, we have engineered the G-QoSm framework to fulfill resource requests with QoS specifications, perform advance reservations of resources, generate QoS agreements, and execute services based on prenegotiated QoS agreements. In the rest of the paper, we focus on the third elementthe commitment level of resource availabilityas we discuss the implementation of G-QoSm and provide initial experience results.

## 7. Implementation and Results

Our test-bed resources included two Linux-based computers: one with a 1.8 GHz Pentium processor and 256 MB of memory, for the service consumer; the other a 1.2 GHz Pentium processor and 512 MB of memory, for the service provider. All machines were connected through a fast Local Area Network (LAN) using Ethernet. Deployed on these machines were the Globus Toolkit version 3 OGSI service container, the Globus Toolkit version 2, and the Java CoG Kit. We experimented with the nanostructure application using two different approaches: 1)

with a QoS handler through the Java CoG Kit, and 2) with a GT2 handler through the Java CoG Kit.

### 7.1. Time-Domain Example

In this section we show results for a nanostructures image analysis task, based on a sample electron diffraction using up to 900 input images. We used a time-domain strategy for resource allocation, with the entire computer node reserved for the application; multiple jobs were submitted to the reserved node, but only one was executed, i.e. the job which had previously made a reservation.

We conducted two sets of runs: 1) job submission based on QoS, and 2) standard job submission based on GT2 GRAM. Each set consisted of eight runs to analyze 25 images, 50 images, 75 images and 90 images. For the first run, job submission is based on QoS properties, the four sets of images were processed in two modes; i) parallel, i.e. submitting the entire set of images to the Grid node for processing at the same time, and ii) sequential, i.e. submitting one image at a time to the Grid node for processing. Similarly for the second run, job submission makes use of GT2 GRAM – in this instance we also used the parallel and sequential modes. Figures 8, 9, 10, and 11 show the performance results, with number of images and time taken to process that number of images for each run.

Experiment results displayed in Figures 8 and 9 make use of the QoS approach, and show that the time taken to process the images, in both parallel and sequential modes, is less than in the GT2 approach. This result is expected as the reservation mechanism employed in this time-domain strategy is to reserve the entire processing power of the Grid node for the QoS-based application, and this prevents other processes from using the processing power while the reservation holds.

Experiment results displayed in Figures 10 and 11 show the use of the GT2 approach, indicating that the time taken to process images, in both parallel and sequential modes, is more than in the QoS approach. The reason is that multiple processing loads were applied to simulate a shared multi-user environment. This background workload generator is used to sort a list of random numbers – upto 10,000 (the actual size of the number of elements in the array is also picked randomly) – using a variety of different sorting algorithms. We also specify a random wait period between each invocation of the random number generator – to simulate the creation of new jobs at unpredictable times. Executing this process adds a variable workload to the existing jobs that are being managed by a CPU. As the GT2 technology does not employ

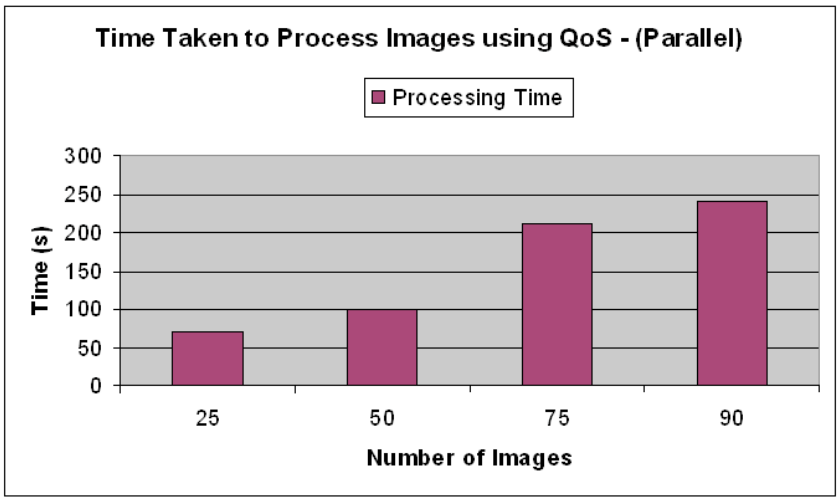


*Figure 8.* QoS-Based Execution – Parallel

a reservation mechanism, other processes could use processing power while the job submitted was being processed.

Figures 12 and 13 are, respectively, results for executing the nanostructure application in GT2 (Best Effort Service) and with QoS (Guaranteed Service), and show the processing time taken to analyze each image.

Figure 12 consists of GT2 (Best Effort) results, indicating that processing time per image generally takes from 10 to 30 seconds. This 20 second variation in the image processing time is quite significant, as this makes the processing pattern unpredictable, and, therefore, unreliable.

Figure 13, using our proposed QoS (Guaranteed) approach, has an execution time per image, ranging from 10 to 12 seconds, except for image number '36', which took approximately 15 seconds. The same image took approximately 37 seconds in Figure 12, based on the GT2 (Best Effort) mechanism, which indicates that image '36' has more processing requirements than the other images. The variation in image processing time with the use of QoS constraints is quite small, which makes the processing pattern consistent and, hence, reliable. From the above results we observe that application processing using the proposed QoS approach provides the following advantages:

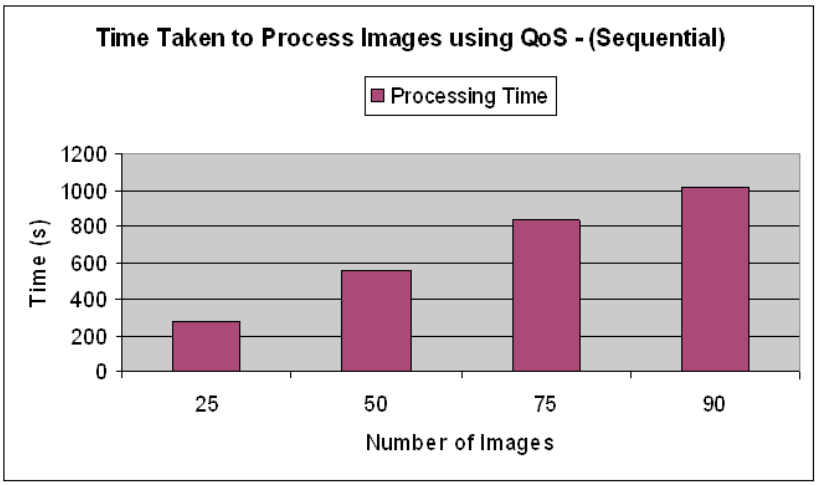


*Figure 9.* QoS-based Execution – Sequential

- The processing of the images has a better performance.
- The time variation in processing each image is about 2 seconds, compared to the GT2 approach of 20 seconds using the same set of images. This difference is quite significant, and the proposed QoS approach is thus more predictable and reliable.

### 7.2. Resource-Domain Example

In this section we show results when the G-QoSm framework was used to allocate CPU resources with a QoS specification using a resource-domain allocation strategy. With this strategy, a slot of CPU time is reserved, and the client application can submit jobs to be executed under fractional reservation constraints. The process is implemented using the Java CoG Kit API to create a task object, and then submitting the created task to the QoS Handler to negotiate the required resources or services. Upon success, a Service Level Agreement is returned for use when claiming a reserved resource in the future.

To evaluate the behavior of the proposed system under heavy load, and observe the effectiveness of the job-submission with QoS constraints, we run two experiments as follows: i) two processes run

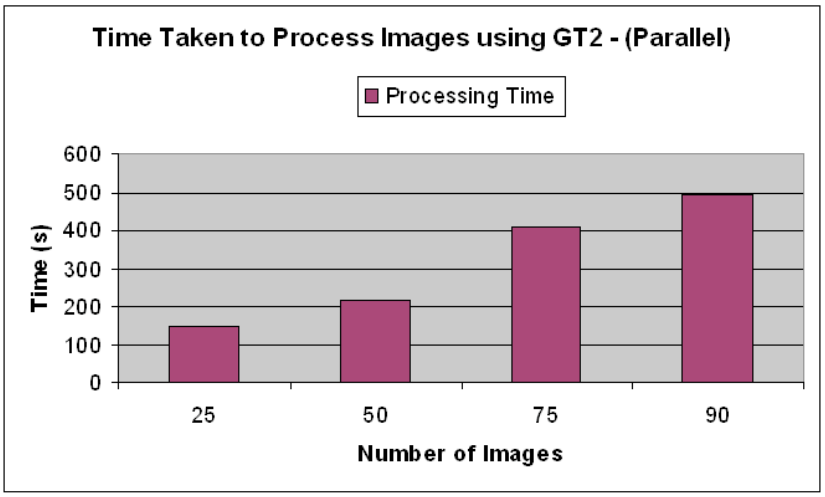


*Figure 10.* Best Effort Execution using GT2 – Parallel

in best-effort mode, i.e. without CPU resources reservation, and ii) a process run in guaranteed mode, i.e. with CPU resource reservation for 60% from time 25s to time 65s. The guaranteed process is run for a specified time frame, while the competing processes (best-effort) were running.

To further study the behavior of our system, and to observe the execution pattern of the guaranteed process, we ran the guaranteed process for a specified time frame, while the competing processes were running. Performance data was taken shortly before the guaranteed process starts, then periodically, until shortly after it completes. Figure 14 plots the execution pattern:

– From time 10s to 25s, two computation-intensive processes are competing for 100% use of the CPU.

– At time 25s the guaranteed process, with a guaranteed CPU usage of 60%, started and lasted until time 65s, due to the previously made reservation.

– From time 65s the two computation-intensive processes are competing again for 100% use of the CPU.

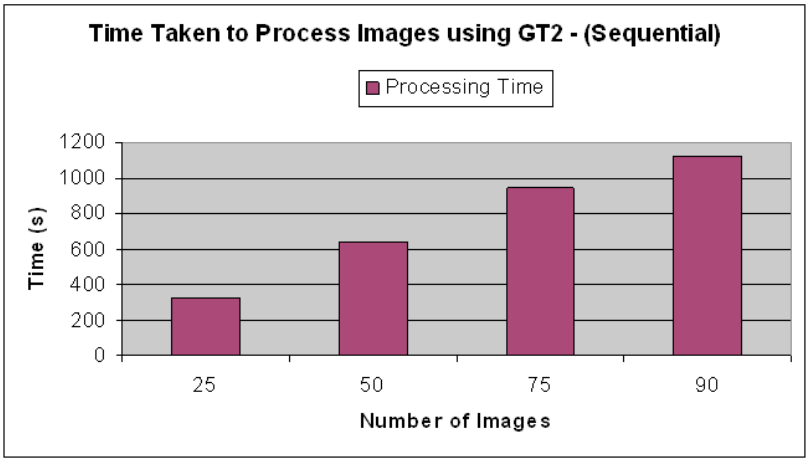


*Figure 11.* Best Effort Execution using GT2 – Sequential

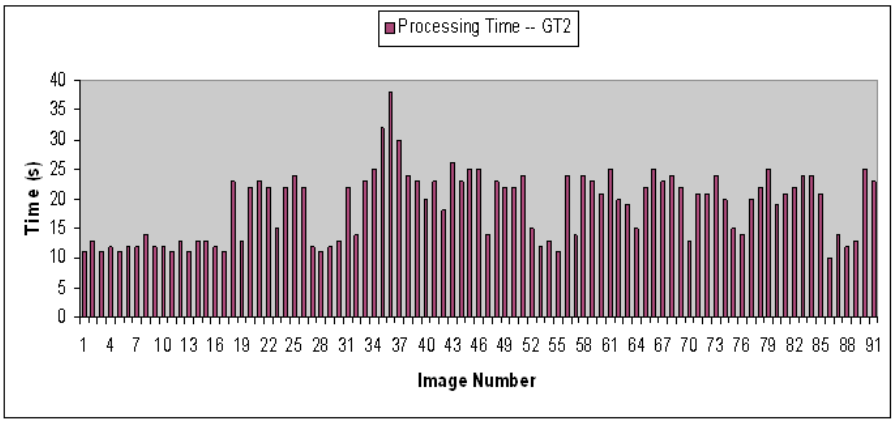


*Figure 12.* The Application using GT2 – Best Effort Service

As a result, it is clear that during the active session of the guaranteed process, it maintained the guaranteed CPU usage of 60%, with the rest of the CPU shared between the other processes. At time 65s, when the guaranteed process completed, the two computation-intensive processes, started to compete for 100% usage.

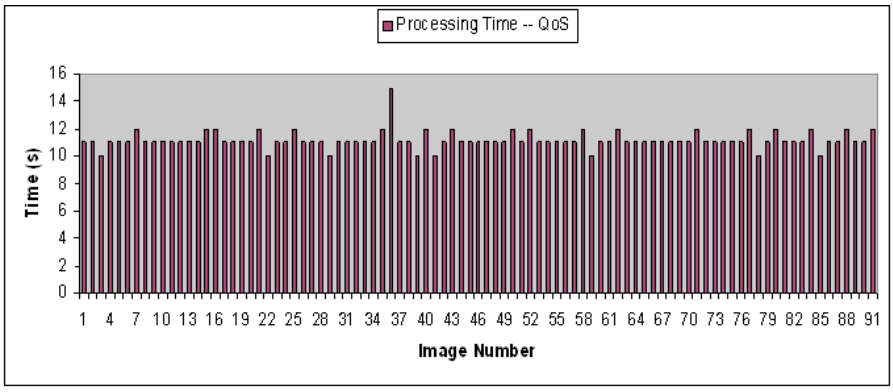


*Figure 13.* The Application using QoS – Guaranteed Service

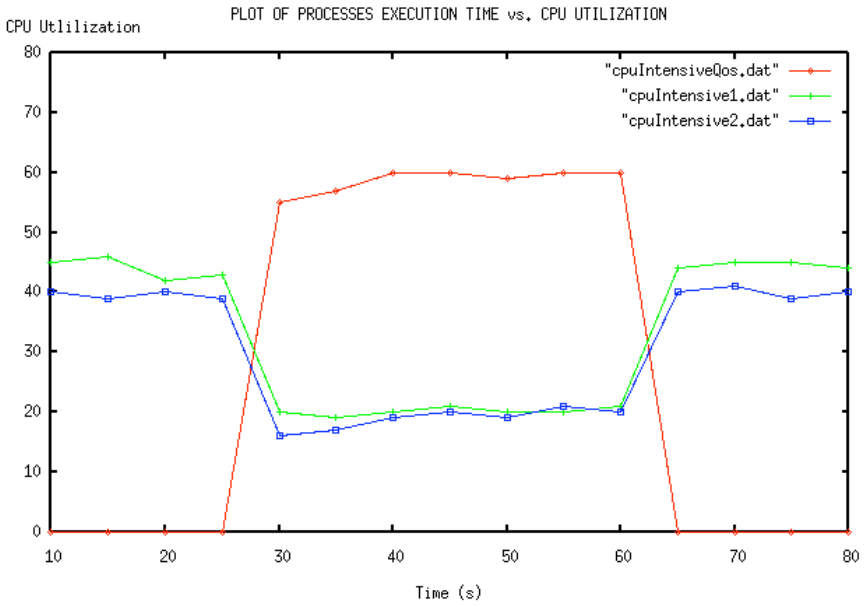


*Figure 14.* Execution of 'Guaranteed' and Competing Processes

## 7.3. QoS Overhead and System Limitation

To further study the proposed system, we conducted two experiments to establish: 1) QoS overhead imposed on job submission, and 2) System limitation in terms of maximum number of requests the system can manage before it fails to respond adequately (essentially to test the scalability). We can specify the limitations of the present system as follows:

1. QoS Overhead: The most apparent overhead the QoS imposes on conventional job-submission is the introduction of the QoS negotiation and advance/immediate resource reservation. This overhead is realized when the client/application submits a request to the QoS service for resource reservation, with QoS constraints and subsequent resource allocation. The QoS service undertakes resources discovery and reservation and presents the user with a Service Level Agreement (SLA) for use when claiming the service. To measure the overhead imposed in this process of negotiating the SLA, we monitored a client generating, at various times, 1,000 requests for the QoS service. The time taken was recorded – from the client initiating the request to the QoS service, until the request was acknowledged. We observed that the time taken to acknowledge QoS requests ranges from a *best-case* of 50 ms to a *worst-case* of 200 ms. The acknowledgement time depends on how busy the QoS service is, and on the network connecting the consumer and the provider. *50* to *200 ms* is not a significant period compared to the actual time the QoS session is reserved for – usually of the order of minutes or even hours, and this overhead is, thus, negligible.

Table1: Requests Acknowledged and Time Taken

| *Requests* | *Period*(*minutes*) |
|---|---|
| 3,857 | 4:01 |
| 3,594 | 5:34 |
| 3,606 | 6:40 |
| 3,596 | 8:23 |
| 3,603 | 9:22 |

2. System Limitation: Grid middleware should be scalable, especially Grid systems that deal with a large number of users and resources. A test was therefore conducted to find out how scalable our proposed system is, in terms of maintaining QoS requests. A large number of requests were issued from clients, at different times over the network to the QoS service. It was observed that the QoS service cannot accept more requests after some 3,600 request, when denial of service takes place. Table 1 shows the number of requests accepted and time taken. The system was analyzed to identify the cause of this limitation, which was found to be that the prototype

system employed a reservation table, which contains related information about the reservations, agreements and SLAs. This table was built and stored in primary memory, and when denial of service occurred the memory was found to be 80% full – this figure was obtained by running the `top` Unix utility to determine free memory. The denial of service could also have arisen due to the process table becoming full as more requests were sent to the server. To overcome this constraint, we will build the reservation table in a disk file, to accommodate a higher number of requests. Swapping overhead to retrieve data from this file, however, may render this approach unusable as the number of requests increase.

## 8. Conclusion and Future Work

We discuss the importance of QoS to support Grid computing applications. QoS criteria in Grid computing are viewed from three perspectives: networking, computation, and storage media. We also outline general requirements for QoS management in the context of service Grids.

A Grid QoS resource management architecture, called G-QoSm, is described. This architecture overcomes some of the limitations of earlier efforts in the Grid community, such as GARA. The development of G-QoSm benefits from our experience in designing the Java CoG Kit, which uses convenient abstractions to integrate QoS capabilities and is easily ported to the Globus Toolkit version 2 and 3.

A G-QoSm prototype is used within a nanomaterial structures application, and is used as an illustrative example. Our architecture currently includes a set of components that abstract the use of QoS for the non-programmer. We emphasize that these components are critical if the Grid is to gain widespread acceptance in real applications. The current set of components must be augmented and their utility demonstrated to convince and encourage new users to utilize Grid computing resources.

We intend to continue our research in Grid resource management in accordance with the Global Grid Forum Grid Resource Agreement and Allocation Protocol working group – especially recent work towards the WS-agreement standard. We believe, however, that a resource agreement and allocation protocol is just a small fraction of the work necessary to enable full use of QoS properties in Grids. Investigation into related areas, such as real-time resource allocation strategies and capacity planning remain important research areas to support

collaborative applications – such as those that require computational steering support.

## Acknowledgment

This work was supported by the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38. DARPA, DOE, and NSF support Globus Alliance research and development. The Java CoG Kit Project is supported by DOE SciDAC and NSF Alliance.

## References

R. Al-Ali, K. Amin, G. von Laszeswski, O. Rana, and D. Walker. An OGSA-Based Quality of Service Framework. In *Proceedings of the Second International Workshop on Grid and Cooperative Computing (GCC2003)*, Shanghai, China, 2003.

R. Al-Ali, A. Hafid, O. Rana, and D. Walker. An Approach for QoS Adaptation in Service-Oriented Grids. *Concurrency and Computation: Practice and Experience Journal*, 16(5):401–412, 2004.

R. Al-Ali, O. Rana, D. Walker, S. Jha, and S. Sohail. G-QoSM: Grid Service Discovery using QoS Properties. *Computing and Informatics Journal, Special Issue on Grid Computing*, 21(4):363–382, 2002.

K. Amin, M. Hategan, G. von Laszeswski, and N. Zaluzec. Abstracting the Grid. In *Proceedings of the 12-th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP 2004)*, A Coruna, Spain, 2004.

J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci, and H. Remzi. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, 2002.

G. Bochmann and A. Hafid. Some Principles for Quality of Service Management. Technical report, Universite de Montreal, 1996.

L. Burchard, M. Hovestadt, O. Kao, A. Keller, and B. Linnert. The Virtual Resource Manager: An Architecture for SLA-aware Resource Management. In *Proceedings of IEEE CCGrid 2004*, Chicago, US, 2004 (to appear)

H. Chu and K Nahrstedt. A CPU Service Classes for Multimedia Applications. In *IEEE Multimedia Systems '99*, 1999.

K. Czajkowski and A. Dan and J. Rofrano and S. Tuecke and and M. Xu Agreement-based Grid Service Management (OGSI-Agreement). Global Grid Forum, GRAAP-WG Author Contribution Draft, June 2003.

K. Czajkowski and S. Fitzgerald and I. Foster and C. Kesselman Grid Information Services for Distributed Resource Sharing. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), 2001.

K. Czajkowski and I. Foster and C. Kesselman and V. Sander and S. Tuecke SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing, 2002.

I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservation and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid:An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, January 2002.

I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3), 2002.

V. Deora, J. Shao, W. A. Gray, and N. J. Fiddian. A Quality of Service Management Framework Based on User Expectations. First International Conference on Service Oriented Computing (ICSOC), Trento, Italy, December 2003.

The Globus Alliance. Web Page http://www.globus.org/.

M. Karsten, N. Berier, L. Wolf, and R. Steinmetz. A policy-based service specification for resource reservation in advance. In *International Conference on Computer Communications (ICCC'99)*, 1999.

K. Keahey and K. Motawi The Taming of the Grid: Virtual Application Service. Argonne National Laboratory Technical Memorandum No. 262, May 2003.

J. MacLaren. Advance reservations: State of the Art. GGF GRAAP-WG, See Web Site at: http://www.fz-juelich.de/zam/RD/coop/ggf/graap/graap-wg.html, Last visited: August 2003.

A. Oguz, A. T. Campbell, M. E. Kounavis, and R. F. Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Pesronal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, 5(4), 1998.

A. Roy. *End-to-End Quality of Service for High-End Applications*. PhD thesis, The University of Chicago, August 2001.

Gregor von Laszewski, Mei-Hui Su, Joseph A. Insley, Ian Foster, John Bresnahan, Carl Kesselman, Marcus Thiebaux, Mark L. Rivers, Steve Wang, Brian Tieman, and Ian McNulty. Real-Time Analysis, Visualization, and Steering of Microtomography Experiments at Photon Sources. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, 22-24 March 1999.

A. Sahai and S. Graupner and V. Machiraju and A. Moorsel. (Specifying and Monitoring G)uarantees in Commercial Grids through SLA. Proceedings of the 3rd IEEE/ACM CCGrid2003, 2003.

Shaleeza Sohail, Khoi Ba Pham, Richmond Nguyen and Sanjay Jha. Bandwidth Broker Implementation- Circa-Complete and Integrable. Proceedings of $7^{th}$ International Symposium on Digital Signal Processing and Communication Systems, Coolangata, Australia, 2003

A. ShaikhAli and O. Rana and R. Al-Ali and D. Walker UDDIe: An Extended Registry for Web Services. Proceedings of Workshop on Service Oriented Computing: Models, Architectures and Applications, 2003.

G. von Laszewski and I. Foster and J. Gawor and P. Lane A Java Commodity Grid Kit. Concurrency and Computation: Practice and Experience, 13(8-9), 2001.

G. von Laszewski and P. Wagstrom. Tools andEnvironments for Parallel and Distributed Computing. *Series on Parallel and Distributed Computing*. Wiley, 2004, ch. Gestalt of the Grid, pp. 149187.

S Jarvis, D Spooner, H Keung, J Dyson, L Zhao and G Nudd. Performance-based Middleware Services for Grid Computing. Proceedings of the $12^{th}$ IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), 2003.

N. Zaluzec. Argonne National Laboratory TPM/AAEM Collaboratory. See Web Site at: http://tpm.amc.anl.gov/.