Quality of Service Management in Service-oriented Grids

Rashid J. Al-Ali

B.S., University of the Pacific, USA, 1992
M.S., The George Washington University, USA, 1997

Thesis submitted for the degree of
Doctor of Philosophy

School of Computer Science
Cardiff University, Cardiff, UK, October 2005

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Dedication

For my Family, Friends and Colleagues

Acknowledgements

I must, in the first instance, thank Dr Omer Rana, my first supervisor, for his many suggestions and support during this research, and Professor David Walker, my second supervisor, for his help and support; without their constant advice and support this thesis would not be complete.

I am also grateful to Dr Gregor von Laszewski of the Argonne National Laboratory (ANL), for his help in gaining access to the ANL Grid infrastructure, testing our work on a scientific application and contributing to the Java CoG Kit project. Thanks also to Kaizar Amin from ANL for useful discussions on integrating the QoS work into the Java CoG Kit, and to Mihael Hategan, also of ANL, for support in integrating the scientific application.

Special thanks to Dr. Abdelhakim Hafid, Dr. Karim Djimam, Professor Peter Dew, Dr. Raj Kumar, Shaleeza Sohail, Dr. Sanjay Jha, Dr. Sander Volker, Dr. Klara Nahrstedt, Ali ShaikhAli, Dr. Simone Ludwig, Dr. Steven Lynden, Hema Arora and Karthika Arunachalam for their support, and to members of the School of Computer Science at Cardiff – my appreciation for a stimulating work environment.

I must acknowledge, with thanks, those who have funded me throughout this research project; The Qatar government for sponsoring me throughout the research period, and Cardiff University for covering participation at a number of conferences, with an acknowledgement also to the Java CoG Kit project for appreciated support.

I am especially grateful to my mother, for her support and prayers, and to my lovely family, my wife Madiha and my children, Ahmad, Abdullah, Fatema and Noora, for their constant support and patience during my absence for much of the past four years.

Abstract

Grid computing provides a robust paradigm for aggregating disparate resources in a secure and controlled environment. The emerging grid infrastructure gives rise to a class of scientific applications and services in support of collaborative and distributed resource-sharing requirements, as part of *teleimmersion*, *visualization* and *simulation* services. Because such applications operate in a collaborative mode, data must be stored, processed and delivered in a timely manner.

Such classes of applications have collaborative and distributed resource-sharing requirements, and have stringent real-time constraints and *quality-of-service* (QoS) requirements. A QoS management approach is therefore essential to orchestrate and guarantee the interaction among such applications in a distributed computing environment. Grid architectures require an underpinning of QoS support to manage complex computation-intensive and data-intensive applications, as current grid middleware solutions lack QoS provision. QoS guarantees in the grid context have, however, not been given the importance they merit. To enhance its functionality, a computational grid must be overlaid with an advanced QoS architecture to best execute those applications with real-time constraints.

This thesis reports on the design and implementation of a software framework, called Grid QoS Management (G-QoSm). G-QoSm incorporates a new QoS management model and provides a service-oriented QoS management approach that supports the Open Grid Service Architecture. Its novel features include grid-service discovery based on QoS attributes, immediate and advance resource reservation, service execution with QoS constraints, and techniques for QoS adaptation to compensate for resource degradation, and to optimise resource allocation while maintaining a service level agreement.

The benefits of G-QoSm are demonstrated by prototype test-beds that integrate scientific grid applications and simulate grid data-transfer applications. Results show that the grid application and the data-transfer simulation have better performance when used with the proposed QoS approach. QoS abstractions are presented for building QoS-aware applications, in the context of service-oriented grids. These abstractions are application programming interfaces to facilitate application developers utilising the proposed QoS management solution.

Quality of Service Management in Service-oriented Grids

Contents

Figures

Tables

Algorithms

Notation

ANL	Argonne National Laboratory
API	application programming interface
BB	bandwidth broker
BE	best effort
BSLA	bind SLA
COPS	Common Open Policy Service
COPS-PR	COPS for provisioning
CPU	central processing unit
DiffServ	differentiated services
DMM	distributed multimedia
DSCP	DiffServ Code Point
DSQDP	Domain-specific Query and Discovery Protocol
DSRT	Dynamic Soft Real-time
EF	expedited forwarding
FSQDP	Full Search Query and Discovery Protocol
GARA	General-purpose Architecture for Reservation and Allocation
GGF	Global Grid Forum
G-QoSm	Grid QoS Management
GRAAP	Grid Resource Allocation Agreement Protocol
GRAM	Globus Resource Allocation Manager
GRIA	Grid Resource for Industrial Applications
GT*x*	Globus Toolkit version *x*
GUI	graphical user interface
H-FSC	Hierarchical Fair Service Curve
IETF	Internet Engineering Task Force
IntServ	Integrated Services
IP	Internet Protocol
Java CoG Kit	Java Commodity Grid Kit
MM	multimedia
MG	Manufacturing Grid
NAFUR	Negotiation Approach with Future Reservation
NFC	National Fusion Collaborator
NRM	Network Resource Manager
NRS	Network Resource Scheduler
OGSA	Open Grid Services Architecture

OGSI	Open Grid Service Infrastructure
PDP	policy decision point
PEP	policy enforcement point
PIB	policy information base
QGS	QoS Grid Service
QoS	*quality-of-service*
RAA	resource allocation answer
RAR	resource allocation request
RB	Resource Broker
RFC	request for comment
RM	Resource Manager
RSLA	resource SLA
RTARM	Real-time Adaptive Resource Manager
RTT	round trip time
SDS	Service Discovery Service
SIBBS	Simple Inter-domain BB Signalling protocol
SLA	service level agreement
SLA-ID	SLA identifier
SLS	service level specification
SNAP	Service Negotiation and Acquisition Protocol
SOA	service-oriented architecture
SP	service provider
SRT	Soft Real-time
TAST	Timely Adaptive State Tree
TCP	Transmission Control Protocol
TSLA	task SLA
UDDI	Universal Description Discovery and Integration
UDDIe	extended UDDI
UDP	User Datagram Protocol
VAS	Virtual Application Service
VDHA	Virtual Dynamic Hierarchical Architecture
VO	virtual organization
WS	Web Service
WSA	Web Service Agreement
WSDL	Web Services Description Language
WSRF	Web Services Resource Framework
XML	e**X**tensible **M**arkup **L**anguage

Quality of Service Management in Service-oriented Grids

Chapter 1 ~ Introduction

1.0 Background

Grid computing, which can be viewed as *'coordinated resource sharing within multi-institutional organizations'* (Foster *et al.* 2001), originally focused on large-scale sharing of distributed resources, scientific applications and the achievement of high performance (von Laszewski *et al.* 2003). A *grid architecture* integrates diverse network environments, with widely varying resource and security characteristics, into a *virtual organization* (VO). Computational grids offer high-performance computing facilities that can be exploited by advanced scientific and commercial applications. Such facilities provide computational resources with high storage capacities and/or processing power to execute applications with special resource requirements, such as data-intensive and computation-intensive applications.

Until recently, research on grids focused on designing and building middleware that address the core problem of grids, such as the management of resources and services in a distributed environment (Argonne, 2004). Such services include resource management, security and data management; services fundamental to grids, as they deal with accessing resources in distributed computing environments which exist in multiple domains. Argonne National Laboratory (ANL) has developed an open-source grid middleware, called Globus, which has become the *de facto* grid middleware for research, and also, more recently, for production purposes.

Although the grid community has produced a number of other systems – Legion (The Legion Project, 2004) and NetSolve (NetSolve, 2004) to name a few – many areas of the grid concept remain to be investigated. Promising research directions include resource management, security and networking, particularly with the use of *Web Services* (WS) technologies, which offer a new approach to building and utilising

services in distributed computing environments. Some advantages of this new approach are: i) loose coupling in application-to-application interaction, or application to data sources, via Internet technology, and ii) a protocol based on using eXtensible Markup Language (XML) message encoding.

1.1 Service-oriented Architecture

A Service-oriented Architecture (SOA) is essentially a collection of inter-communicating services passing and exchanging data, and co-ordinating some activities. Services are self-contained, and well-defined, software entities, each with an interface and behaviour i.e. service capability. Services exchange messages with applications or other services; for example in Web Services (WS) technology, these messages are encoded as eXtensible Markup Language (XML) messages and are encapsulated into Simple Object Access Protocol (SOAP) envelopes; with services thus 'language' independent and designed to support inter-operability (Taylor, 2005).

Web Services is a technology in a SOA for connecting services, with services connected through WS, and 'service' the endpoint of a connection, i.e. basically a software capability. In WS services can be advertised by a service provider, to a service repository, such as the Universal Description Discovery and Integration (UDDI), through a process called '*publish*'. A Web Services standard, the Web Services Description Language (WSDL), is used to advertise service-related information, such as the service interface, i.e. how a client can invoke a set of pre-defined operations on another service. A service can further be discovered by a service requestor, sometimes called service consumer, through a process called '*find*', which, essentially, searches the service repository, such as the UDDI, to locate suitable services, described in the service WSDL description. A service can be invoked through a process called '*bind*', i.e. making use of the service capability by sending a *request* to the remote service and receiving a *response* over the network; essentially an exchange of XML messages.

The Open Grid Service Architecture (OGSA) is an architecture-specifying grid system based on Web Services concepts and technologies (Argonne, 2004). OGSA presents grid functionalities as a collection of services called 'grid services'. Grid

services are essentially Web services with additional features such as stateful, lifetime management and notification support. In OGSA all resources and applications are presented as grid services, with one noticeable feature that grid services are manageable, and, unlike Web services, grid services can be created, destroyed or even monitored. OGSA defines a common standard for grid-based applications, and developed an Open Grid Service Infrastructure (OGSI) standard to provide technical specifications for grid services. OGSA has recently produced the Web Services Resource Framework (WSRF) (Czajkowski *et al.* 2004) standard to overcome some limitations of the OGSI, such as specifications for *stateful* services.

In this thesis the proposal for the design and implementation of a QoS management system is envisioned as a grid service conforming to the OGSA standard. Such a QoS grid service delivers QoS management functionality to applications or other grid services. The stateful feature of grid services, defined by the OGSA standard, is essential for the proposed QoS grid service, as this grid service deals with applications and other services to provision QoS assurances, referenced by an agreement called a Service Level Agreement (SLA). SLAs should be stored, and accessed when applications want to utilise the services with QoS provision, as specified in a SLA. Any request for services with QoS provision goes through a validation process which verifies the requesting application has, indeed, a provisioned QoS level specified in a SLA. SLA information should be associated with QoS grid service; such an association can be delivered by the stateful feature specified by the OGSA for grid services. (A further discussion on SLAs is given in Chapter 3.)

1.2 Quality of Service

Quality-of-service (QoS) issues have been explored in various contexts: network, multimedia and, more recently, resource management, as discussed further in Chapter 2. The work described here focuses on QoS issues in resource management for distributed computing in *service-oriented architectures* (SOAs), and, in particular, in the context of the Open Grid Services Architecture (OGSA) (Foster *et al.* 2002). QoS can be defined as a measure of performance for certain service quality, where the service could be networking, multimedia or certain resources e.g. processors – sometimes called central processing units (CPU) in the following

Chapters. The QoS is normally specified in a set of parameters describing the desired service: for example, a networking service is described by a group of parameter, including *bandwidth*, *delay*, *jitter* and *packet-loss rate*.

Grid services conform to certain specifications, are self-contained and provide well defined interfaces. Grid services are hosted in grid resources and infrastructures; and connectivity is maintained among resources via dedicated high-speed networks. A well-established grid infrastructure facilitates constant resource connectivity, resource monitoring and fault tolerance. Hence some basic level of QoS is provided by the committed members of a VO, based on their pre-agreed grid policies and their dedication to collaboration. Nevertheless, the complexities involved in critical grid applications require *guaranteed* QoS assurances beyond those provided by a basic grid infrastructure, such as critical applications with real-time requirements. Because of the increasing sophistication of grid applications (TeraGrid, 2001), such as those with real-time constraints, QoS provision becomes an inherent requirement in a grid architecture. A modern SOA requires advanced management to provide QoS assurances of meeting such application requirements.

QoS depends on the context in which it is addressed. For example, QoS in multimedia deals with the presentation quality of multimedia documents, while network QoS deals with communication-link characteristics, such as bandwidth and delay. QoS management, for the purpose of this thesis, is defined as *all activities, from resource selection and allocation through to resource release, intended to ensure a set of qualitative and quantitative attribute values*. Examples of qualitative QoS attributes include service reliability and user satisfaction, while examples of quantitative QoS attributes include network bandwidth, processor performance and storage capacity, which implies a certain capacity of disk storage for application use.

Overlaying an advanced QoS framework on existing grid architectures allows the support of complex QoS requirements. The work presented in this thesis is the design and implementation of a software framework called Grid QoS Management (G-QoSm) that provides QoS functionality in SOAs. G-QoSm supports recent standardisation efforts by the Global Grid Forum (GGF) (The Global Grid Forum,

2004) and is compatible with the OGSA specification. Important features of G-QoSm are:

- It is based on the concept of a *service level agreement* (SLA) that contains service-related details and agreement terms. A SLA comprises the contract document between a user and a QoS management entity, which specifies the services and quality the user should expect.
- It employs a service-selection mechanism in the service-discovery process to select the most appropriate service, based on user-supplied service requirements.
- It supports advance resource reservation to guarantee resource availability when needed.
- It incorporates techniques for QoS adaptation to compensate for resource QoS degradation during the active phase of a QoS session.

The process of establishing SLAs, in the context of G-QoSm, shares many similarities with the WS-Agreement (WSA) standard (Andrieux *et al.* 2004). For example, in a job submission for a WSA, the provider posts an agreement template, comprising a list of available applications, and the service consumer is required to populate the template with information on the desired application, such as the application name, the number of required processing nodes and other job submission parameters, including, for example, the source of input data. Once the template is returned to the provider, the consumer waits for confirmation, or rejection, of the agreement – if the agreement is rejected, the consumer can try again with different parameters in the agreement template – which basically constitutes a negotiation process. In G-QoSm a similar approach is taken to negotiate and establish SLAs, as outlined in Section 4.2.

The effectiveness of G-QoSm is validated by building two prototype test-beds; the first incorporating a scientific grid application and the second the simulation of a grid data transfer application. The first prototype demonstrates computation QoS and the second demonstrates network QoS. Performance results demonstrate the benefits of the proposed QoS-based deployment.

1.3 Research Methodology and Hypothesis

Hypothesis: QoS management in a SOA can provide a guaranteed, reliable and consistent service-execution mechanism.

A new architecture for QoS management is proposed, which addresses the questions:

- How can a QoS management system be presented as a Web Services (WS), in the context of SOAs, where users and applications interact through standard WS protocols?
- How can a typical service-oriented application utilise and benefit from use of such a QoS management approach?
- What performance gains can be obtained by an application using such a QoS management system in a SOA?

The hypothesis is verified by comparing the performance of the G-QoSm prototypes to a grid middleware system without QoS management support, based on two measures:

- **Computation QoS:** defined as guaranteeing a certain percentage of processor capacity for an application in a shared processor system, or guaranteeing a processor, or a number of processors, for an application's exclusive use in a multiprocessor system (Roy, 2001). In this instance, the computation QoS measures the time taken to complete a QoS-aware application process while other applications utilise system resources.
- **Network QoS:** defined as guaranteeing a certain quality level of a network link between two end points, where the link characteristics include delay, jitter, packet loss rate and bandwidth. In this instance, the network QoS is the ability of a QoS-aware application to maintain a promised rate of data transfer while other applications utilise system resources.

1.4 Novel Contributions of the Thesis

The thesis is motivated by the desire and need to develop a QoS management system for SOAs and particularly for service-oriented grids. It envisions that the proposed

approach would be of great benefit for the Globus toolkit *ecosystem* (Liming, 2004). The Globus toolkit ecosystem is based around the OGSA concept, and outlines grid architectures for various types of applications, such as computation-intensive, data-intensive and distributed collaborations. This thesis proposes a QoS management system which could be utilised in such architectures to guarantee a required QoS level for applications accessing grid resources.

The novel aspect of this thesis is the proposal of a QoS management system, called G-QoSm, to provide QoS functionality for grid resources, such as computation and networks. The G-QoSm prototype is designed and implemented in the context of OGSA as a grid service within Globus Toolkit version 3 (GT3). Additional contributions to research on grid and QoS management, raised in the development of the new QoS management system, include:

- Development of an abstraction for QoS management in SOAs. The abstraction employs a utility model for cost optimisation; depending on whether the cost for executing a service is calculated by a client or a provider, a user may optimise this cost from different perspectives. Given a particular quality level, a user may be interested in identifying a set of resources that can offer the quality at a minimum cost. Alternatively, a user may be interested in maximising the revenue that could be obtained by selecting from available resources.
- The description of a novel protocol for agreement-based QoS negotiation, which establishes a SLA between a service consumer and a provider.
- New resource selection and *resource domain* and *time domain* resource allocation strategies based on QoS properties: resource domain allocates a certain percentage capacity for a shared resource and is suitable for applications that require limited resources, whereas time domain allocates the entire resource capacity for an application, based on exclusive use, and is suitable for applications that require high-performance resources.
- A new technique for advance resource reservation in grids, for single and/or multiple resources. Reservation of multiple resources is of particular importance in grid systems as normally grid applications require more than a single resource to be simultaneously allocated, also referred to as co-allocation.

- QoS adaptation mechanisms that compensate for QoS degradation and maintain agreed-on SLAs.

1.5 Thesis Outline

- ***Chapter 2 ~ Literature Review,*** surveys the background areas of research related to the main ideas presented in the thesis. These main ideas are resource discovery in distributed systems, resource reservation for QoS-aware systems, literature on SLAs, and related works on the concept of QoS adaptation.
- ***Chapter 3 ~ A Model for Quality-of-Service Provision,*** presents a new agreement-based abstraction for QoS management in SOAs, and discusses the main components of the model.
- ***Chapter 4 ~ Framework Design,*** presents the design for G-QoSm, based on the model presented in Chapter 3, and discusses the modularity of the G-QoSm design.
- ***Chapter 5 ~ The Prototype,*** discusses implementation issues for the proposed QoS management system, describes the prototype implementation, and discusses how a grid application can utilise the proposed G-QoSm system.
- ***Chapter 6 ~ Validation,*** presents performance results of the G-QoSm prototype, based on experiments undertaken in collaboration with ANL and Cardiff University.
    - Work at ANL integrated an image-processing grid application based on *nano materials*, with this application demonstrating the need for computation QoS (Al-Ali *et al.* 2004a/2004b).
    - Work at Cardiff University demonstrates network QoS for data-transfer applications (Al-Ali *et al.* 2004d).
- ***Chapter 7 ~ Conclusion,*** presents a summary of the results, discusses the outcome of the work and makes recommendations for further study.

Chapter 2 ~ Literature Review

2.0 Synopsis

In this Chapter literature on QoS management is surveyed; the concept of QoS is defined and the activities and functions undertaken during a QoS session are presented. QoS issues with reference to grid computing are introduced, and the requirements for a QoS-aware grid-resource management system are identified. Existing research projects dealing with QoS in distributed computing are discussed, and the concepts in these projects are compared to the research presented in this thesis. Work related to functions essential for QoS management is reviewed, including: a) resource discovery; b) resource reservation; c) Service Level Agreements; and d) QoS adaptation. In addition to these four functions this Chapter includes a review of network QoS for grid applications. A number of QoS management systems are also reviewed and compared to the proposed G-QoSm system.

2.1 Quality of Service

The concept of QoS was first used in the network community (Aurrecoechea *et al.* 1995). In this context, network QoS specifically deals with providing certain quality levels for network link characteristics between two points, with these characteristics expressed as delay, jitter, throughput and packet loss rate:

- *Delay:* Time it takes a packet to travel from a sender to a receiver;
- *Jitter:* Variation in the delay of packets taking the same route;
- *Throughput:* Rate at which packets travel through the network;
- *Packet-loss rate:* Rate at which packets are dropped, lost or corrupted.

To manage these network parameters, certain network elements – network routers or network traffic-control entities, such as Linux-based routers – are modified to support QoS models, such as Differentiated Services (Blake *et al.* 1998; Xiao and Ni, 1999), or changes are made at the application end-points to control how packets are transmitted, based on feedback from the receiver. The first of these – modifying

network elements – is usually undertaken at the network level; a very effective mechanism as it controls the physical network link. The alternative approach is an application-level solution, where feedback on network performance is used to control the rate at which data is transmitted from the sender.

The QoS concept was next introduced in resource management applications, and particularly in distributed multimedia (DMM) (Campbell *et al.* 1993; Narhstedt and Smith, 1995; Bochmann and Hafid, 1996). QoS in resource management deals with the issue of providing certain service qualities to applications, whereas in the multimedia community QoS issues are concerned with providing a client with an acceptable level of presentation quality when accessing a multimedia document. This level of quality includes support for QoS at the network level, which forms a connection between client and server, in addition to providing certain guarantees for resources on the server side, comprising computing (processor performance) – to process and dispatch, for example, multimedia frames at specific rates.

QoS was introduced into the grid computing community prior to 2004. The Globus Alliance (Argonne, 2004) discusses the concept of the General-purpose Architecture for Reservation and Allocation (GARA) (Foster *et al.* 1999). In the context of grid computing, some effort has been expended in introducing a specialised network QoS to support grid applications; exploiting ideas and concepts from the networking community (Bhatti *et al.* 2003). Recently, with the introduction of the OGSA concept, QoS provision has been introduced in the context of service-oriented grids (Al-Ali *et al.* 2002a). The QoS work presented in this thesis benefits from concepts related to QoS investigated in different communities, such as networking and DMM. QoS in SOAs, and specifically in OGSA, is the theme of this research.

Although there is extensive research on QoS in various communities, there is no standardisation; although some communities have working groups setting up architecture and specifications for QoS. For example, in the context of the networking community, the Internet Engineering Task Force (IETF) has released a request for comment (RFC), describing a network QoS architecture based on differentiated service (Blake *et al.* 1998). Similarly, the GGF has a working group called Grid Resource Allocation Agreement Protocol (GRAAP), which is involved in

a number of issues related to QoS (MacLaren, 2003) and is establishing standards for resource description, reservation and agreements. The GRAAP working group primarily addresses the protocol to reserve and allocate resources in grid environments.

QoS has no standard definition and is therefore, normally, defined according to the context in which it is used. For instance, Jarvis *et al.* (2003) define QoS as a representation of *user-side* service (i.e. user perception) based on deadlines assigned to tasks, while Roy (2001) defines QoS as guaranteeing the availability of specific resource characteristics in a shared resources environment, such as processor performance or network bandwidth.

QoS provision in a shared resources environment is essential, as, with any finite set of resources, the resources are, eventually, fully occupied and no further clients or applications can utilise the resources. To overcome this problem, either a QoS management system, which can support reservation mechanisms and admission control procedures to access the resources, must be provided, or the finite set of resources must be increased to accommodate requirements for all expected client or application needs. The second solution is not usually acceptable as it is virtually impossible to provide access to unlimited resources, and provision of QoS management functionality is normally more efficient and cost effective. This thesis focuses primarily on proposing a design, and building a QoS management system.

2.1.1 – QoS Management Functions

A QoS session has three main phases: (1) the ***establishment*** phase; (2) the ***active*** phase; and (3) the ***clearing*** phase (Hafid and Bochmann, 1998). Each phase has QoS functions, as shown in Figure 2.1.

During the establishment phase a client states their QoS specifications, and the QoS management entity undertakes service and resource discovery based on QoS properties negotiated with the client (Al-Ali *et al.* 2003d).

During the active phase, additional activities such as resource allocation, based on previously-reserved resources, QoS monitoring, accounting, adaptation and possibly

re-negotiation may take place. Some activities in this QoS management phase may be repeated a number of times; for example, a re-negotiation may trigger resource allocation being re-applied, and similarly for adaptation when allocated resources fall below the agreed-on specifications.

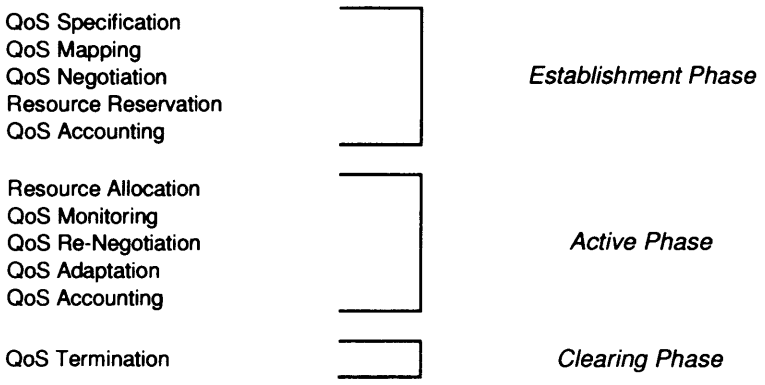

Figure 2.1: QoS Management Functions

The clearing phase occurs when the QoS session is terminated, due to a resource reservation ending, a SLA violation, or service completion, which frees resources for use by other clients. To detect a SLA violation, the QoS levels – i.e. resource specifications – must be monitored. For example, Baker and Smith (2003) propose a grid resource monitoring system called GridRM – a generic resource monitoring framework capable of providing a client/application with resource data. This data can be used by the QoS management entity during the active phase.

This thesis is mainly concerned with four aspects of QoS management in a grid context:
(i) Resource Discovery: concerned with discovering and selecting grid resources based on QoS properties, such as resource specifications, during the establishment phase of QoS management.
(ii) Resource Reservation: part of the establishment phase, and an important function in providing resource access guarantees.
(iii) Service Level Agreements: cover the entire spectrum of QoS management. These agreements, negotiated in the establishment phase, are used in both the active and clearing phases, and may be re-negotiated during the active phase.

(iv) QoS Adaptation: triggered primarily during the active phase, this process is concerned with resource allocation and adaptation and is meant to compensate for QoS degradation.

When studied in the context of grid computing, QoS research differ from other communities in two main areas: (a) the nature of available resources; and (b) the simultaneous allocation of resources that span multiple administrative domains. QoS in grid computing usually deals with more than one type of resource because of the co-allocation requirements of many grid applications, whereas most other communities, such as networking, normally only deal with one type of resource. Grid resources include computation nodes, networks, storage devices and specialised instruments; normally found in more than one administrative domain. This domain-spanning is the main distinguishing feature of a grid system. In resource management terminology, this can be viewed as coordinating multiple resource access – which may be simultaneous – spanning multiple domains, in scientific, or commercial, applications.

The nature of QoS provision depends on the nature of the resources involved. For example, processor QoS depends on whether a processor is being used as a shared or an exclusive access resource (Roy, 2001). With processor sharing, an application can specify that it requires a certain percentage of processor capacity over a specific time period. In a multiprocessor system an application can also specify exclusive access to a number of processors over a specific time period.

Similarly, storage QoS concerns access to storage devices such as disks. In this context, QoS is characterised by bandwidth and storage capacity. Bandwidth is the rate of data transfer between a storage device and an application program. 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 concurrently used. The number and types of parallel I/O channels available between the processor and the storage media are significant parameters in specifying storage QoS. Storage capacity is the volume of storage space an application can use, during its execution, for writing data.

2.2 QoS in Grid Computing

It would be convenient for a grid application to specify its QoS requirements in the form of a single (virtual) resource, necessary to run the application, comprising computing, storage and networking resources, and the period over which the resource is required. Such a resource may, in practice, involve the aggregation of a number of distinct grid resources to achieve the desired outcome.

A grid application usually submits its requirements to a grid resource management service that schedules jobs as resources become available. Each resource provider supports a resource manager that receives requests from external applications. Certain applications, such as real-time and collaborative applications, need to obtain results within strict deadlines, and cannot always wait for resources to become available. Others require multiple resources to be simultaneously allocated, with no strict deadlines. For such applications, it is often necessary to reserve grid resources for a specific time; in advance, or immediately. Guaranteeing resource availability for an application's execution is highly desirable, indeed, it is required if grid services are to handle complex scientific and business applications that need resources distributed over multiple administrative domains.

Taylor (2005) sees QoS in grid systems as a key parameter, and negotiating SLAs to address QoS requirements as essential. Taylor categorises QoS in grid systems into three types:

- **None:** verifying that QoS is not supported; similar to *best effort* support.
- **Soft:** implying QoS can be specified, but the resource management system cannot provide guarantees. This is the most common form of QoS in grid computing.
- **Hard:** meaning that all nodes on the grid support, and guarantee, QoS.

In the following sections, requirements for a QoS-aware grid resource management system are presented and the extent to which current QoS systems meet such requirements is discussed.

2.2.1 – Requirements

A grid resource management system should address the following requirements:

- **Resource Reservation:** should support mechanisms for immediate or advance resource reservation. Advance reservation is particularly important for resources shared in multi-user environments.
- **Reservation Policy:** should support mechanisms for resource owners to enforce policies governing when, how and who can reserve their resources. For reservation flexibility the policy mechanism should be decoupled from the reservation mechanism (Karsten *et al.* 1999).
- **Protocol for Negotiating SLAs:** should assure clients of the resource configuration expected during the service session. Such assurance can be given in an agreement document, such as a SLA. Creation of such a document requires a negotiation mechanism so service consumers and providers can negotiate SLA terms, such as service starting time and resource specifications.
- **Security:** should prevent malicious users from penetrating or altering data repositories holding information about reservations, policies and agreements. In addition to a secure channel between an application and the grid resources being used, a security infrastructure providing support for authentication and access control is also required.
- **Simplicity:** should have as simple a design as is reasonable, requiring minimal or no changes to existing infrastructure.
- **Scalability:** should be scalable to a large number of entities. This is especially true since grids are expected to be open and dynamic, with resources and users joining and leaving in a non-deterministic manner.
- **Resource Co-allocation:** should be able to simultaneously deal with multiple resources, as a typical grid application requires different types of resources to be allocated concurrently.

2.2.2 – QoS in Grids

In grid computing, QoS management must provide the required access to computing resources in multiple domains. Unlike multimedia and network QoS, grid QoS requires a global information service (Fitzgerald *et al.* 1997; Cjazkowski *et al.* 2001), which is a central virtual resource, consisting of a number of replicated information

services, to have global information readily available on the status of resources. This is essential, as the grid consists of diverse resources distributed over multiple domains. Such a service can be interrogated by an application to determine which resources it can use. Because grid QoS deals with concurrent service sessions, SLAs are essential to specify resource configurations for each service with these configurations encoded in the SLA as parameters. Subsequently, each parameter can be monitored to ensure SLA conformance.

SLAs encode particular resource requirements for an application as SLA *elements*, which represent SLA terms, for example, the required network bandwidth or required processor performance. These elements can be verified against resource capabilities a specific owner can provide. Such SLAs, between a service consumer and service providers, can be expressed using first-order logic.

Relatively few systems have been developed that provide QoS support for grid applications; with examples including GARA (Foster *et al.* 1999), the Virtual Application Service (VAS) (Keahey and Motawi, 2003), and the GRIA project (GRIA, 2004). The Grid Resource for Industrial Applications (GRIA) project targets industrial applications and attempts to provide end-to-end performance and availability estimation, with efficient mapping of workloads to resources. It uses techniques such as *workload estimation* and resource *capacity estimation* to accomplish QoS-based performance. A notable feature of the GRIA is that it does not provide absolute guarantees that a resource will be available to run the required job at a specific time, but does allow a client to specify requirements, and agrees on what should happen if these requirements are not met. This approach to QoS management does not engender a high degree of confidence that a job will be executed, or that results will be collected on time. As discussed in Section 2.2.3 the GARA and VAS systems share many similarities with the work in this thesis.

Other QoS efforts in the grid community are mostly attempts to manage network properties for grid applications. Examples include The Network Resource Scheduler (NRS) project (Bhatti *et al.* 2003). The objective of the NRS is to provide the users, and applications, with a means to request network capacity allocation, with immediate or advance reservation. This network resource allocation provides QoS

guarantees over grid domains, such network QoS utilising the *differentiated services* (DiffServ) concept (Blake *et al.* 1998). Section 2.7 discusses the network QoS for grid applications.

2.2.3 – Discussion: GARA and VAS

Although networking support is important, GARA and VAS are designed not only to provide network QoS but also other types of QoS, such as processor performance. The following sections discuss these two systems and highlight their differences.

2.2.3.1 General-purpose Architecture for Reservation and Allocation (GARA)

GARA is the best known framework for supporting QoS in computational grids, and provides the ability of specifying end-to-end QoS requirements. Its advance reservation service treats various types of resources uniformly such as networks, computation and storage, and provides a guarantee that an application initiating a reservation will receive a specific QoS from the resource manager. This is made possible by employing specialised resource managers to support QoS guarantees. GARA also provides an application programming interface (API) to *create*, *modify*, *bind* and *cancel* reservation requests.

Although GARA has gained popularity in the grid community, it has limitations in coping with current application requirements and technologies. For example:

- GARA does not operate in an OGSA context, and OGSA-enabled applications cannot use it directly. Grid computing increasingly relies on WS technologies, and many current grid middleware systems are moving towards WS standards (Foster *et al.* 2002) and placing greater importance on the Web Services Resource Framework (WSRF) (Czajkowski *et al.* 2004).
- GARA does not support protocols for agreements, or the establishment of SLAs, which are essential requirement for dealing with resources spanning multiple administrative domains. The GGF is working on standardising agreement protocols, which address resource negotiation with QoS specifications, through the GRAAP working group (Czajkowski *et al.* 2003).

- GARA does not support a QoS adaptation feature for computational resources, although QoS monitoring and adaptation during an active QoS session is one of the important mechanisms in providing quality guarantees (Al-Ali *et al.* 2004c).
- Although GARA is, in principle, portable, it is based on earlier versions of Globus (Version 2.2 and earlier), and is not currently maintained.

2.2.3.2 Virtual Application Service (VAS)

Keahey and Motawi (2003) propose the VAS architecture for managing QoS in computational grids. VAS is a grid service with interfaces for negotiating QoS levels and service requests. A key objective is to support real-time services with QoS provision. A client submits a request to VAS for immediate or advance reservation, supplying only time constraints. Application modelling information associated with every service allows the system to compute the feasibility of satisfying such time constraints. If feasible, the modelling information, such as execution times and hardware resource data, allows the system to determine the computational resources required to support the request, and to reserve a specific processor capacity. A SLA is then presented to the user based on these parameters.

VAS is a deadline-driven system, in which a client specifies only the time constraints (start time and deadline time) and VAS computes the feasibility of meeting this deadline. This approach is ambitious but is, in reality, limited to a set of predefined services. This view is supported by the fact that VAS is designed for a specific application domain called the National Fusion Collaborator (NFC) (National Fusion Collaboratory, 2005).

2.3 Resource Discovery

Resource discovery is the process of locating resources in a distributed computing environment, where a resource can be of any type, including computing nodes, networks and storage devices (Foster *et al.* 2002). A number of techniques have been introduced to solve the discovery problem. For example, Ludwig and van Santen (2002) use ontology-based descriptions to enhance the matchmaking process of service discovery in grids. Lican *et al.* (2003), investigating algorithms for service discovery, propose the Virtual Dynamic Hierarchical Architecture (VDHA). They

claim that VDHA supports scalable, autonomous, efficient, reliable and quick response, and propose two service-discovery algorithms: (1) Full Search Query and Discovery Protocol (FSQDP), and (2) Domain-specific Query and Discovery Protocol (DSQDP). Service discovery based on VDHA is fully decentralised and unrelated to service-description languages, because it uses local agents of nodes to match the services, and can scale to a large number of services; scalability, in the context of distributed computing, is highly desirable because of the potential for service growth.

Rana *et al.* (2001) also utilise agents to solve the discovery problem; for example, they propose a decentralised approach to resource management and discovery, based on a community of interacting software agents, unlike the solution proposed in this thesis using a centralised discovery system.

Mechanisms for service discovery based on QoS properties in grids, DMM applications and network services have recently been explored. In grids, several such mechanisms are based on the Universal Description Discovery and Integration (UDDI) project (UDDI, 2004). The myGrid project (Moreau *et al.* 2002) involves middleware intended to provide a toolbox for biologists and bio-informaticians performing workflow-based *in silico* experiments, so as to automate the management of such workflow. The concepts of QoS registration for service instances are explored in the *service directory* of the myGrid project though the use of UDDI-M an extension to the standard UDDI service directory approach that supports service metadata storage via a tunnelling technique that ties the metadata store to the original UDDI directory (Dialani *et al.* 2002). Search mechanisms based on QoS properties, a desirable feature for any QoS-based discovery system, are not supported in UDDI-M.

The GARA project (Foster *et al.* 1999), does not address specifications of QoS associated with a particular service and the service concept is not supported by GARA. Service discovery based on QoS has also been explored in the context of grids, with a demonstration of how a feedback capability on service performance can improve QoS. The Wide-area Discovery Framework (Xu *et al.* 2001), is a hierarchical architecture of three elements, *service clients*, *service providers* and *discovery servers*, which work together to constitute a wide-area distributed-system

service directory management. This service directory management is enhanced to provide better query responsiveness and QoS awareness. Feedback, in this context, means that, during a service session, a software component monitors QoS levels and generates the numerically-average QoS level observed. The definition of this QoS level is highly service-specific, i.e. dependant on the type of service being considered. This project targets queries which must traverse a number of discovery services in a hierarchical fashion.

In the context of DMM applications, Madja *et al.* (1998) propose a data model for QoS management on the Web. Their data model is a set of QoS characteristics for multimedia audio/video documents. This data may be stored in a database, as text files, or as an extension of HTML tags. A client specifies the desired quality of the multimedia document and the QoS manager accesses the multimedia document's metadata to negotiate the requirements identified by the client. This work, however, is limited to multimedia documents, and not general enough to support the concept of services.

In the context of network services, the Service Discovery Service (SDS) provides an architecture consisting of clients, services and SDS servers (Czerwinski *et al.* 1999). This architecture includes a number of interesting features such as security, scalability and the notion of a capability manager. The capability manager has an access control list to indicate which users have the right to access which service. In SDS, a client searches for services based on their capability rather than the client's QoS requirements. In an agreement-based system, the client/application must be able to specify their QoS requirements, so a negotiation can take place and an agreement can be reached.

The Darwin system (Chandra *et al.* 1998) is a service-oriented resource management system capable of managing requests for complex network services with QoS support. A request is entered into the system by the user in the form of a task graph. A resource manager locates suitable resources to perform the requested tasks with the optionally-specified QoS requirements. The resource manager is responsible for creating a *hierarchical grouping*, which consists of a structure of the network flows, with their QoS specifications and the IP addresses of the nodes. This hierarchical

grouping tree is passed to the designated network resource manager(s) for the allocation process.

Darwin has four main components: (1) a high-level resource allocation mechanism and a resource broker named Xena, to perform global allocation of resources using domain knowledge to support optimisations; (2) runtime resource managers and Java *control delegates*, which support service-specific adaptation for network resources; (3) a hierarchical scheduling mechanism, the Hierarchical Fair Service Curve (H-FSC) scheduler, which enables each participating resource to specify its own policy; and (4) a signalling protocol, named Beagle, which provides an interface between an abstract view of the network and the real physical network.

The concept of a service in Darwin is quite restrictive, with its primary focus on network resources. Support for generic services such as computation, storage or other services is limited. In the Darwin system, Xena does not employ a general resource discovery protocol, rather, it offers a mechanism through which services can register their availability and capabilities, i.e. a simple publish-subscribe mechanism. This allows Xena to build a coarse-grained database of available resources.

The systems surveyed in this section do not address the issues of QoS criteria specified within a service interface, such as service capability and resource specifications needed to run the service properly. Such criteria are particularly important when a service is distributed on a number of hosts, or when there are multiple service providers who can provide the same service, but with different QoS capabilities. Much emphasis has been placed in previous work on building service discovery mechanisms that attempt to minimise response time. Generally, such approaches utilise a hierarchical scheme to aggregate and propagate network statistics (Yemini *et al.* 1991; Lin and Stadler, 2001). Although such approaches are adopted in the context of service discovery (Hass *et al.* 2001; Xu *et al.* 2001), issues arising as a consequence of using QoS properties have not been adequately addressed.

2.4 Resource Reservation

A reservation can be viewed as a promise from a QoS manager to a client of expected resources with a certain capability to be available during a certain time. Advance resource reservation is defined as a *possibly limited or restricted delegation, of a particular resource capability over a defined time interval, obtained by the requester from the resource owner through a negotiation process* (MacLaren, 2003). A resource reservation can be categorised either as an *advance* reservation or as an *immediate* (also called *on-demand*) reservation, which can be for a specified, or indefinite, duration.

Indefinite reservation is undesirable as it introduces blockages that can result in a waste of unused resources. But an important feature of reservation, of particular importance to grid computing, is support for co-reservation. Immediate and advance reservations are used in a wide variety of systems, mostly in networking, communication and distributed applications, including DMM applications. A number of systems with advance/immediate reservation features have been proposed in the networking and DMM communities, whereas few systems are proposed in the context of grids.

Negotiation Approach with Future Reservation (NAFUR) is a QoS negotiation system with advance reservation support in the context of DMM applications (Hafid *et al.* 1998). It computes the QoS that can be supported at the time of a service request or at a certain later predetermined time. If a multimedia service with a certain QoS cannot be supported at the time of a request, NAFUR computes the earliest time at which the service can be supported with that specific QoS. This *counter offer* reservation feature is quite desirable, but NAFUR is restricted to DMM applications.

In Kim and Nahrstedt (2000), a resource broker (RB) model in the context of middleware for DMM applications is proposed with the following design goals:

(1) Advance and immediate reservations;

(2) A new admission control scheme based on a Timely Adaptive State Tree (TAST); and

(3) The processing of brokerage requests for resource reservation, modification, allocation and release.

The admission control, based on TAST, is used to make advance reservation decisions, with TAST based on an algorithm that provides QoS suggestions to users. These suggestions, provided when the original QoS request is rejected due to resource unavailability, can be to reduce reservation duration, to degrade QoS or to select a different start time. The use of TAST to make admission control decisions is a notable feature of this model, which is useful as it provides suggestions when the original request cannot be granted, as opposed to a YES/NO response. The approach is however limited to DMM applications.

In Karsten *et al.* (1999), advance reservation is formalised in the context of networking systems, and the fundamental problem of admission control associated with resource reservation is introduced. Based on a literature review, the authors conclude that no earlier approach is sufficiently flexible to cover all potential needs of all users. Their solution is to separate the issue into technical and policy specifications, supported by a generic reservation service description and a corresponding policy layer. This combination improves the flexibility of advance reservation. Although this advance reservation approach is intended for networking systems, and deals with only one type of resource, it can be generalised for multiple resources.

None of these research projects address advance reservation in the context of service-oriented architectures. Nevertheless, the GGF GRAAP working group has produced a 'state of the art' document laying down properties for resource reservation in grids (MacLaren, 2003). None of the systems reviewed address the concept of co-reservation for advance/immediate resource reservation; such co-reservation is of particular importance for grid applications, as they often simultaneously utilise multiple grid resources. For example; the GARA framework, in the context of grid computing, does not provide co-reservation support – the reservation of multiple resources in a single request. The approaches reviewed focus mostly on providing alternative reservation suggestions where an original reservation request cannot be granted, i.e. counter-proposal reservations. Such approaches are useful but are usually limited to a predefined set of services. In the context of grid computing

however, applications deal with multiple types of resources, and services dynamically join, and leave, the grid.

2.5 Service Level Agreements

A SLA, in the context of grid services, is a contract between a service provider and a consumer (Czajkowski *et al.* 2003). A SLA contains *general* and *service-specific* elements. General elements, a part of every SLA, are independent of the service and include, for example, a contract validity period, as well as penalties for SLA violations. For example, service-specific, or technical elements include, service execution requirements in terms of resource capability specifications and, perhaps, performance requirements.

Bhoj *et al.* (1998) present a Web-based SLA management for network services in a federated system, including a framework for contract verification with a visual interface for contract compliance reports. Nguyen *et al.* (2002) propose a protocol for negotiating service-level specifications (SLSs) as the technical elements of a SLA for intra- and inter-domain network services, based on the Common Open Policy Service (COPS), called COPS-SLS.

Pard *et al.* (2001) discuss the management and control of SLAs for multimedia Internet services using a *utility model*; a mathematical model designed to capture the management and control aspects of SLAs. This particular utility model has been used in micro-economics theory, and is defined, in this context, as *the satisfaction obtained from a service provider for the consumed system and network resources.* The aim of this approach is to address management and control aspect of the QoS levels while utilising the system and network resources efficiently. This is achieved through the concepts of:

(1) a *quality profile* that specifies the quality performances of customers, i.e. a set of acceptable operating qualities for a service;

(2) *quality-to-resource mapping*, which maps the qualities specified in the SLA to the available resources;

(3) *resource constraints* – the sum of all resources allocated to customers, which cannot exceed the total available resources, and

(4) a *utility model* that maps the customer's operating quality to a utility value. This work is useful as it associates resource operating qualities and utility values. It is however limited to the multimedia Internet services domain.

The Service Negotiation and Acquisition Protocol (SNAP), introduced by Czajkowski *et al.* (2002), is a resource management model for negotiating resources in distributed systems such as grids. SNAP incorporates three types of SLA: the task SLA (TSLA), the resource SLA (RSLA), and the bind SLA (BSLA). A TSLA describes a task to be executed while the RSLA describes the resources needed. The BSLA provides an association between the resources from an RSLA and the task in a TSLA. The protocol requires a resource management entity to guarantee resource capability and provide resource provision, i.e. to enforce the RSLA.

In a *manufacturing grid* (MG), resources are classified by their function and type, and encapsulated as grid services (Shi *et al.* 2003). When a client application requests a manufacturing task with QoS specifications, a designated resource management entity generates a workflow schedule, consisting of subtasks, services and resources needed, encoded into a SLA. The generated SLA includes a description of the workflow, with each task of the workflow defined as a grid service with its QoS specifications. This work is useful as an approach to QoS-based workflow, but the applicability is restricted to the manufacturing application domain and operates on predefined services.

Sahai *et al.* (2003) describe a SLA management entity for supporting QoS in the context of commercial grids. In commercial grids businesses are bound by commitments specified in SLAs, and monitoring and accountability therefore becomes important. The SLA management entity exists within OGSA – with its own set of protocols for manageability and assurance. SLA management also needs interfaces to the service factory, registration and discovery service, for finding resources based on QoS requirements, and interfaces with a notification service to notify impacted parties on SLA status. The authors also describe a formal language for SLA specification. Although interesting, this work is preliminary and its general applicability is not altogether clear.

Burchard *et al.* (2004) propose SLAs for negotiating service execution parameters between resource managers. SLA management is achieved via a *virtual resource manager* that enables interaction among a number of schedulers on different clusters. The virtual resource manager acts as a coordinator to aggregate SLAs negotiated with different sub-systems.

Sahai *et al.* (2001) explore application-level QoS. Their work focuses on relating client QoS criteria with business metrics such as revenue. According to Sahai *et al.* SLAs between two parties should be based on the business transactions conducted between them and a transaction focus can then be used to identify criteria that are important, for both clients and service providers. Hence, the QoS criteria for a client are motivated by metrics such as the performance, reliability and availability of a service, whereas a service provider would prefer to differentiate between transactions, provide throughput guarantees, support load balancing across available resources, and support smooth degradation on overload. Basing their argument on these attributes, Sahai *et al.* propose services with high priorities should be provided with a high degree of resource replication to support particular QoS requirements, which would allow a service provider to establish specific performance guarantees for individual transactions. This work encodes application-level QoS criteria into SLAs, and uses the business transaction focus to guarantee SLA compliance.

In this survey the concept of the SLA is explored from various contexts, and is used to encode technical specifications, as for DiffServ services in the networking community (Nguyen *et al.* 2002), such as bandwidth, delay and other parameters, to characterise the networking link. SLAs are also used to encode business terms, and to realise loss/revenue in terms of QoS, as in Sahai *et al.* (2001). SLAs in SOA should extend the traditional concept of SLAs in the networking community to include other QoS parameters specific to SOAs, such as resources needed to run a service and expected response time. With this extension QoS parameters can be realised in SLAs and a client or application can request services with specific levels of quality.

2.6 QoS Adaptation

A *QoS adaptation*, as defined here is used to enable the dynamic adjustment of application behaviour based on changes in a pre-defined SLA. This adjustment can occur when the SLA is violated – i.e. the QoS specified in the SLA has been degraded – or adaptation is necessary to optimise resource allocation during a QoS session (Al-Ali *et al.* 2004c). QoS adaptation can be seen as a reaction by a resource manager to compensate for a resource shortage, such as when QoS has been degraded, optimising resource utility by admitting more requests to share the available resource, while maintaining agreed-on quality levels. Adaptation is particularly useful when workload, or network traffic, changes unpredictably during an active QoS session.

QoS adaptation is also defined as *'the alteration of an application's behaviour or interface in response to arbitrary context changes'* (Henricksen and Indulska, 2001). It has been explored in various contexts such as communication networks, DMM applications, real-time systems and Web browsers. For example, Mobiware, developed at Columbia University (Oguz *et al.* 1998), is a toolkit that supports adaptation at the network level. Mobiware provides programmable network objects that can be manipulated to provide applications with a desired QoS. Applications specify their QoS requirements using an API, in the form of a *utility function* and an *adaptation policy*. The utility function expresses the desired application requirements with different levels of network bandwidth, while the adaptation policy determines how an application's bandwidth allocation should vary as resource availability changes. This approach primarily focuses on network QoS.

Hafid *et al.* (1996) designed and implemented a QoS manager for negotiation and adaptation in DMM applications. Based on a user profile, the QoS manager considers possible system configurations, called *system offers*, and selects an optimal offer, called a *user offer*. During playback of a multimedia document, if the network or the server becomes congested, thereby lowering presentation quality, the QoS manager dynamically considers another system configuration from the list of system offers. If an alternate system offer is selected and the required resources reserved, the manager automatically changes to the new offer.

Chu and Nahrstedt (1999) designed and implemented the Soft Real-Time (SRT) system for multimedia applications. SRT supports multiple CPU service classes for real-time processes, based on the usage pattern of these processes. They use a concept of *contracts* to specify the CPU service class together with a parameter used to reserve CPU cycles. As the processing time per frame in a multimedia application, changes dynamically for some processes, the contract parameters are adjusted accordingly to reflect the change in the processor usage pattern. SRT provides a *system-initiated adaptation* that can adjust contract parameters for the real-time processes based on their actual processor usage. One noticeable feature of this adaptation scheme is the ability to reserve '*just sufficient*' processor time to execute the required processes.

Foster *et al.* (2000) designed and implemented an adaptive control system prototype for grid computing based on: (1) *actuators* that permit online control; (2) *sensors* that permit monitoring of resource allocations; and (3) *a decision procedure* that allows entities to respond to sensor information by invoking actuators. The prototype was implemented with particular emphasis on network resource usage. For example, a loss-rate sensor might acquire information from a network edge router. The decision procedure obtains information from the loss-rate sensor and adapts the network reservation using the GARA *Create/Modify* reservation request, via a reservation actuator. Although this work uses GARA as the underlying resource manager to create, and modify, reservations, this is limited to providing a network adaptation mechanism.

Cardei *et al.* (2000) describe the Real-time Adaptive Resource Manager (RTARM) for resource management adaptation. RTARM is a general middleware architecture for adaptive management of integrated services, and is targeted at real-time mission-critical distributed applications. RTARM recognises three situations where the QoS for an application may change: (1) QoS reduction when a new application begins; (2) QoS improvement when an application terminates and releases resources; and (3) feedback adaptation. Situations (1) and (2) impose contract changes due to adaptation. Feedback adaptation, conversely, does not impose contract changes but operates as a closed-loop control system, monitoring the delivered QoS, and using the

difference between delivered and desired QoS parameters to adapt to application behaviour. The feedback adaptation is intended to optimise resources, even if the contract specifies more resources, and if the application actually uses fewer resources, only those fewer resources are allocated. This approach of adaptation, i.e. feedback adaptation, is useful as resources are optimised. But the situation can arise where the applications, which have had their QoS reduced due to adaptation and are using fewer resources, can change their usage pattern, and require more resource, but all resources are utilised. Such a problem can arise in this type of adaptation mechanism.

The QoS adaptation systems reviewed here have a number of interesting adaptive techniques; for example, the introduction of a decision procedure in Foster *et al.* (2000) and the use of a closed-loop control system, to utilise '*just sufficient*' resources in Cardei *et al.* (2000). However, none of the systems surveyed are SLA-based adaptive systems, i.e. using an adaptation mechanism to maintain an SLA agreement. A QoS-based system should facilitate the negotiating of SLAs, and then, during an active QoS session, provide adaptation behaviour to maintain SLA compliance when the QoS degrades, and optimise resource utilisation while maintaining the agreed-on quality levels. A SLA-based approach is more practical, and provides a mechanism for a client, or application, to negotiate the quality level of service to be received and, eventually, the level of service to be expected during the active phase.

2.7 Network QoS for Grid Applications

Currently the Internet treats all traffic equally as *best effort* and does not support QoS. IETF has proposed *Integrated Services* (IntServ) and DiffServ architectures (Barden *et al.* 1994; Blake *et al.* 1998). Both these architectures support QoS, with data transfer guarantees on bandwidth, delay and other parameters.

IntServ supports network resource reservation by maintaining per-flow admission control, signalling, classification and scheduling at every router on the transmission path. However, because of its need for *maintenance-of-state* information for a large number of flows through core routers, scalability is a major issue preventing the deployment of IntServ.

DiffServ, in contrast, provides a broad and flexible range of services, while avoiding the need for per-flow state information in core routers. The main goal of DiffServ is to provide a preferred level of service to particular types of network traffic, without increasing overheads in the core routers. To achieve this, it provides an aggregated end-to-end service over a number of separately administered domains. As the inter-domain level, i.e. between two domains, needs a mechanism to exchange critical information about aggregated flows, a Bandwidth Broker (BB) (Teitelbaum *et al.* 1999) was introduced to allocate intra-domain resources and arrange inter-domain agreements.

A BB is a logical entity responsible for managing QoS for network resources in an administrative domain, based on a SLA between two domains, or between a domain and an application. Such a SLA specifies to the forwarding service the volume of traffic the application can receive. Organisational policies can be configured by using the mechanism provided by the BB. On the inter-domain level, the BB is responsible for negotiating QoS parameters and setting up bilateral agreements with neighbouring domains. On the intra-domain level, the BB's responsibility includes the configuration of edge routers, to enforce resource allocation and for admission control. Edge routers can be configured to police, and mark, packets with a DiffServ Code Point (DSCP). Policing ensures the receiving rate does not exceed the agreed rate; if exceeded, depending on the adopted policy, excess packets are either discarded or re-marked for a delayed discard if congestion occurs.

There are recent efforts in the grid community to adopt concepts from the network community and to provide QoS in grid applications. Two significant approaches, both DiffServ-based, are the GARA project (Foster *et al.* 1999) and the NRS project (Bhatti *et al.* 2003).

2.7.1 – GARA Network QoS Support

GARA provides network QoS for grid applications based on a DiffServ architecture. Network QoS in GARA is designed, and built, to work with a specific network router, the Cisco 7507, and uses Cisco's Modular QoS Command-line interface to configure routers as a *policy enforcement point* (PEP) to support DiffServ capability.

In a multi-domain network, i.e. multi-administrative domains, the GARA system must exist in every administrative domain. In making a network reservation, for traffic spanning multiple administrative domains, two issues arise: locating and contacting the GARA system in each domain along the traffic path; and ensuring that the application requesting the reservation has secure access to each GARA system along the path. This introduces manageability limitations, and constraints on the administrative domains where GARA is deployed.

2.7.2 – NRS Network QoS Support

NRS adopts a peer-to-peer model, as it exists in every administrative domain, and it is assumed there is a trust relationship between neighbouring NRSs. NRS uses the DiffServ concept, and therefore every neighbouring NRS has a DiffServ SLA (a SLA related to a network connection). The application requesting network QoS needs only negotiate with the local NRS to establish a local SLA. During the negotiation process the local NRS replicates the request, to all NRSs along the network path, to conduct an admission control check and, subsequently, to establish a SLA. NRS, like GARA, is designed, and built, to only work with Cisco routers and to use Cisco-ISO to configure Cisco's routers as PEPs to support DiffServ. Although NRS has demonstrated its effectiveness in providing DiffServ QoS, it is not clear how a grid application developer would make use of this ability, because the NRS API is not clearly defined. Using NRS also requires the definition of specific network parameters, such as traffic specifications, which requires advanced networking knowledge.

2.8 Summary

Research relevant to the thesis is reviewed. QoS management, in the context of grids, is defined and the different functions of QoS management are discussed. Special attention is given to the four main elements of the thesis: resource discovery, resource reservation, SLAs and adaptation. In addition to the network, QoS, in the context of grid applications, is discussed, and key efforts in grid QoS networking are reviewed. Interesting techniques are reviewed concerning the four main elements of the thesis, and it is shown that there is little effort to provide QoS in SOAs.

Since the QoS management problem was first introduced in the networking community, and subsequently in DMM applications, and recently in the grid community, a particular focus is placed on related work in these areas. The novelty of the thesis lies foremost in introducing a generic, modular QoS management framework for SOAs, and for grids in particular (Al-Ali *et al.* 2002b; Al-Ali *et al.* 2003c). The proposed framework gives service providers a means to publish their services with QoS properties, while the service consumer can search for services based on QoS properties, and execute services on resources with QoS properties.

Chapter 3 ~ A Model for Quality-of-Service Provision introduces a conceptual abstract model for QoS management in service-oriented architectures, with the model's features compared to the work reviewed in this Chapter. The model's most significant aspect is an agreement-based model, i.e. a SLA-driven QoS model.

Chapter 3 ~ A Model for Quality-of-Service Provision

3.0 Background

Certain clients of SOAs are concerned with the quality of a service, its computational and economic costs, and that it is executed promptly and properly, in accordance with their expectations. A QoS mechanism identifies the resources needed to execute a service at a specified service quality level. It is important that the selection of such resources be subject to other constraints, such as the cost associated with service execution on a particular set of resources.

The QoS model presented here (Al-Ali *et al.* 2005) distinguishes between a service provider, a service, a resource and a SLA. A resource is an entity that can be reserved, while a SLA is a contract agreed upon between a client and a service provider during the establishment phase of a QoS session, prior to resource allocation. The problem addressed by the QoS model is how to determine, given multiple types of QoS requests from clients, the optimal resource allocation. This is undertaken with reference to a set of pre-defined criteria to maximise resource utilisation and maintain requested quality levels. The model includes optimisation heuristics to discover such resources.

3.1 Synopsis

Advance resource reservations can be viewed as a way to provide a resource access guarantee, and an assurance from the resource management entity that the reserved resources, with the specifications requested, will be available during the agreed-on period. The model has a mechanism for reserving resources in advance for single and collective (i.e. multiple) resources; the latter is called *co-reservation*. Co-reservation is essentially the ability to reserve multiple resources based on a single request, reserving for example, processor time and network resources to run a simulation. The

need for co-reservation arises in grid computing as resources exist across multiple administrative domains, and grid applications, usually simultaneously, utilise resources from these multiple domains.

The proposed model operates in a distributed computing environment. It is assumed resources are shared, and that, during the active phase of a QoS session, resources may become congested, or even fail, causing resource QoS levels to degrade. A means of compensation for such degradation is essential to maintain an agreed-on SLA. A technique for such QoS adaptation is proposed based on reserving extra resource capacity to compensate for any resource shortage – this extra resource capacity is adaptive in the way it is utilised by the *best effort* users (users with no QoS requirements) when not in use by *guaranteed* users. Section 3.3.3.4 elaborates on this adaptation technique. Aspects of the QoS model include the following:

- It is SLA based;
- It employs a service-selection mechanism during the service-discovery process, based on QoS properties;
- The advance resource reservation mechanism employed guarantees resource allocation with certain QoS levels;
- It incorporates techniques for QoS adaptation, to compensate for QoS degradation during the active phase of a QoS session.

3.2 Quality-of-Service Model

Resource management in distributed systems deals with co-ordinating resource allocation for application execution; possibly for multiple clients in a shared-resource environment.

Resource management for a single application in distributed systems, in its simplest form (Rana *et al.* 2002), consists of:

- Selecting a set of resources for executing tasks generated by the application.
- Mapping the tasks to computational resources.
- Routing data to these computations.
- Ensuring that task and data dependencies between tasks are maintained.

In grid computing, a resource management entity usually interacts directly with the grid *middleware*. Middleware is a layer of software that connects processes on computer nodes connected through a network. An example is Globus, a middleware infrastructure from the Globus Alliance (Argonne, 2004) that provides functionality specific to a grid infrastructure, such as co-allocation of resources, data management, information and security services (Czajkowski *et al.* 1998). Baker *et al.* (2000) identify the functionalities of grid middleware as the core services mentioned above, in addition to QoS and resource reservation. Current grid middleware infrastructures, such as Globus, lack QoS and resource-reservation functionality. The QoS model presented here can be used to enhance grid middleware by incorporating QoS functionality and resource reservation support. Chapters 4 and 5 describe the architecture and implementation of a prototype system, and its interaction with Globus middleware. The enhancements are embodied in the following actions:

1. Service providers publish their services with QoS properties to the registry service.
2. A service request consisting of QoS requirements is submitted by a client application.
3. The QoS system selects a service that best matches the specified QoS constraints.
4. A SLA specifying the negotiated service and resource capabilities is issued by the QoS management system to the client application.
5. Resources required to execute the agreed-on service are reserved for later allocation during the active phase of a QoS session.
6. SLA compliance is assured during the active session by monitoring the QoS levels of the allocated resources. Adaptation techniques are utilised if there is QoS degradation.

Figure 3.1, on the following page illustrates the sequence of activities undertaken by the QoS Model.

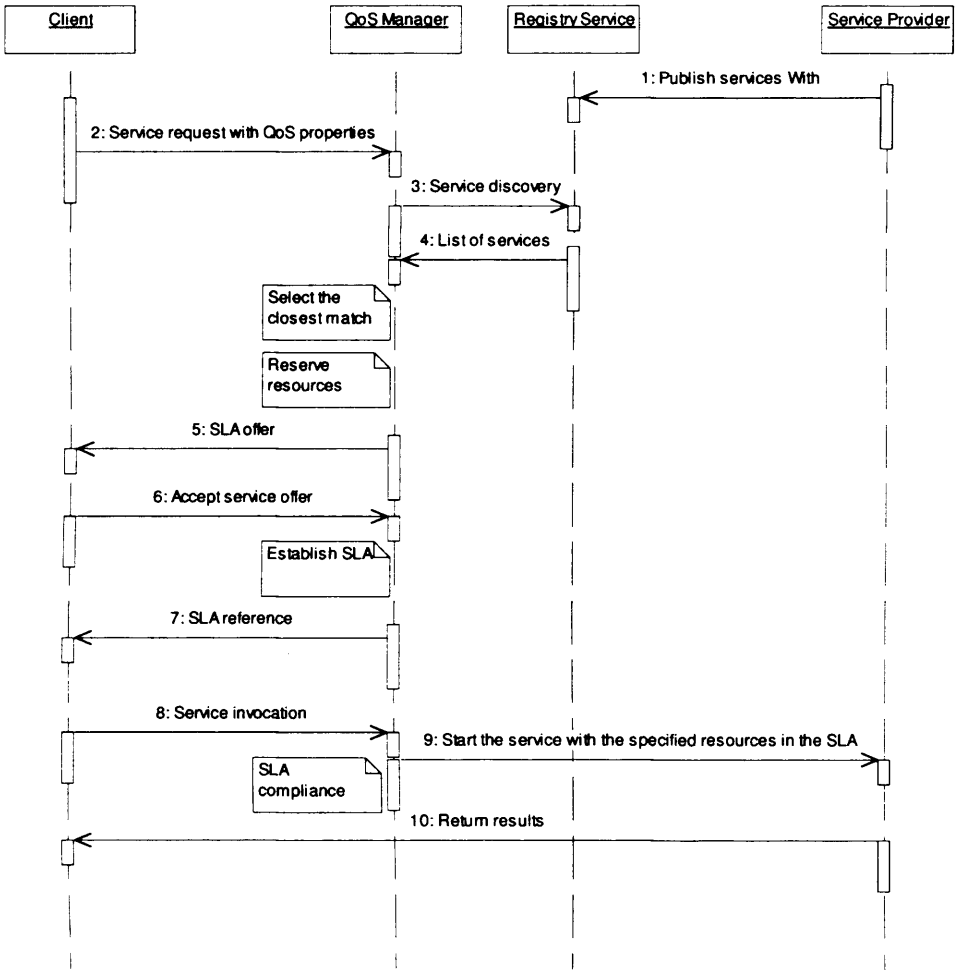

Figure 3.1: Sequence Diagram of Activities undertaken by the QoS Model

Figure 3.2 on the next page illustrates the model and its components. The *QoS Manager* is viewed as a component within the QoS model, and its main objective is to capture requests from the client/application, negotiate SLAs, and enforce SLAs by delivering services with agreed-on levels of quality. The *QoS Registry* is a WS registry system, such as the UDDI system (UDDI, 2004), and is part of the proposed model. The *Service Provider* generates a description of its services, with their QoS properties, such as the required resources and capabilities needed to execute the service, and publishes these to the QoS registry. The Resource Managers (RMs) control a set of resources (*RES*), and interact with the QoS Manager for resource

allocation. *RES* contains subsets of various types of resources, unlike some of the systems reviewed in Chapter 2, which only focus on network resources. The client/application is a consumer that initiates a request for service with QoS constraints.

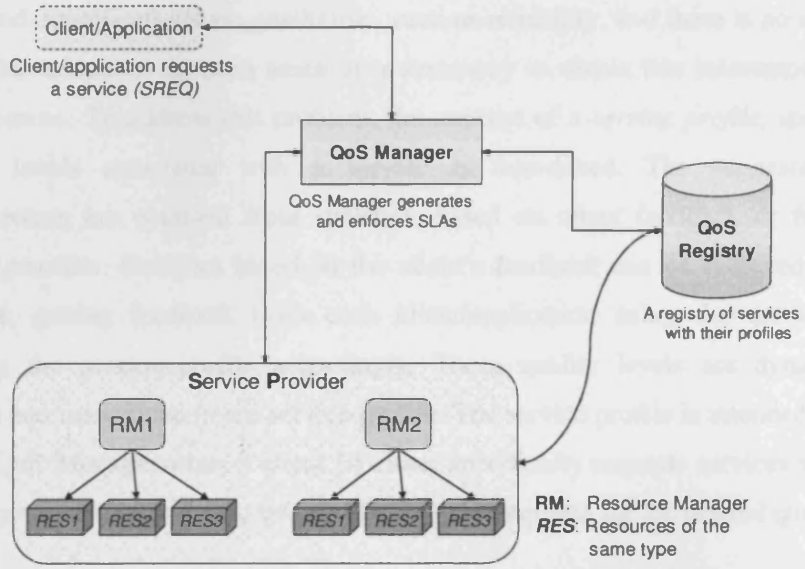

Figure 3.2: The QoS Model Architecture

3.2.1 – Service Request

A client submits a *service request SREQ* to the QoS Manager, specifying a requested service, optional QoS levels, budget constraints and the time interval required for the service. These parameters constitute the client's requirements; the manager searches for services with the specified quality level, finds an appropriate service and starts a negotiation process. If multiple services are found, a selection process is started, or a 'not found' service message is returned to the client/application. A negotiation process in this context means the QoS Manager presents a SLA offer to the client, which the client should approve or reject. If rejecting the offer, the client may submit another *SREQ* with different QoS levels – a process which can be repeated, constituting a negotiation process.

Multiple service requests $SREQ_1$, $SREQ_2$, ..., $SREQ_m$ from m clients may be concurrently submitted. Each requested $SREQ_i$ undergoes a negotiation process in

which the manager considers candidate services and available resources, and selects those most suitable for the client/application's requested QoS specification. Some client requirements, such as reliability, availability, accuracy and response time, are difficult to specify or to measure, and are consequently difficult to capture with monitoring tools such as Netlogger (Gunter *et al.* 2002). Such difficulty arises where some QoS specifications are qualitative, such as reliability, and there is no standard acceptable criterion. In such cases, it is necessary to obtain this information from other sources. To address this problem, the concept of a *service profile*, specifying quality levels associated with a service is introduced. The suggested QoS specifications are obtained from statistics, based on client feedback or from the service provider. Statistics based on the client's feedback can be achieved by, for example, getting feedback from each client/application using the service, and updating the service profile accordingly. These quality levels are dynamically updated and maintained in the service profile. The service profile is intended for use by the QoS Manager when a client [*i*] either specifically requests services with the default profile, or is unable to specify the resource required for the desired quality.

On receiving a service request $SREQ_i$ with the required QoS specifications, whether obtained from client [*i*] or from a service profile, the QoS manager undertakes a *service discovery* process by requesting a list of service matches from the QoS registry, from which it selects the closest match (according to the mechanism described below). The QoS registry, in this context, is characterised by three main features:

(1) *Service properties* – the ability to associate QoS properties with a service through a *publishing process*. This mainly involves QoS information related to the resources required to execute the service, and service utilisation cost;

(2) *Range-based searching* – the ability to search for QoS attributes based on numerical ranges; to give the flexibility of searching for a service with a particular QoS property based on a range. For example 'find services with a required network QoS within the range: $45Mbps>=Net_{Bandwidth}>=155Mbps$'; and

(3) *Service leasing* – the ability to associate a lifetime validity for the service; to publish a service valid for usage during a specific time frame, which is useful

in associating different pricing schemes based on the time of day, such as peak and off-peak hours.

This last feature is motivated by the OGSA service-lifetime management function, such as transient or permanent services (Foster *et al.* 2002). Likewise, in the proposed approach using this soft-state feature, the published services can have associated validity data, so the service provider can set and control this feature. Most of the discovery systems mentioned in Chapter 2 do not support such advanced features, which can add flexibility for QoS-based discovery.

It is assumed that each service request contains numerical values – associating an *importance* level with each QoS attribute – to assist the QoS Manager in making a better selection. The selection is based on the principle of choosing the profile that most closely matches the requested QoS levels, considering the importance level of each QoS attribute stated in the service request.

The selection method is formally described as follows:

Let $SREQ_i = ((q_{i1}, w_{i1}), (q_{i2}, w_{i2}), \ldots, (q_{in}, w_{in}))$ denote a service request from client i, where each q_{ik} is a *resource request* and w_{ik} its associated importance weight, with $r_i = (q_{i1}, q_{i2}, \ldots, q_{in})$ denoting the resource requests from service request $SREQ_i$, and $w_i = (w_{i1}, w_{i2}, \ldots, w_{in})$ denoting the specified importance weights for service request $SREQ_i$. It is important to remember that the service request contains one or more resource requests, i.e. the required resource QoS level.

Each resource request q_{ik} is of the form *(type, value, range)* where *type* is the type of resource requested, such as network bandwidth or processor, *value* is the minimum QoS level acceptable, and *value* + *range* is the highest QoS level the client is willing to pay for. For example; if a request q_{ik} of the form *(type, value, range)* is given as *(bandwidth, 45, 10)* then the request is for *type* = *bandwidth;* with the minimum QoS *value 45* Mbps, and the highest QoS *value 45 + 10 = 55* Mbps. The *type* component of q_{ik} is denoted $type(q_{ik})$, with similar notation for *value* and *range*.

Let $PROF_i = (p_{i1}, p_{i2}, \ldots, p_{is})$ denote the sequence of profiles returned by the QoS registry for service request $SREQ_i$, and each p_{ij} from $PROF_i$ take the form $(q'_{ij1}, q'_{ij2}, \ldots, q'_{ijn})$, *where* q'_{ijk} satisfies q_{ik} in r_i.

A profile resource request $q'_{ijk} = (type', value', range')$ satisfies a service-request resource request $q_{ik} = (type, value, range)$ if $type' = type$, $value' \geq value$, and $value' + range' \leq value + range$.

To define the difference between profile $p_{ij} = (q'_{ij1}, q'_{ij2}, \ldots, q'_{ijn})$ and resource request vector $r_i = (q_{i1}, q_{i2}, \ldots, q_{in})$ then
$r_i - p_{ij} = (((value(q_{i1})+range(q_{i1})) - (value(q'_{ij1})+range(q'_{ij1}))), \ \ldots\ , ((value(q_{in})+range(q_{in})) - (value(q'_{ijn})+range(q'_{ijn})))$.

Assuming two vectors, v and w, then the *norm* of a vector v with respect to vector w can be defined as: $\|v\|_w = \sqrt{\sum_{k=1}^{n} w_k (v_k)^2}$.

The profile that most closely matches the resource requests in r_i is taken to be the profile p_{ij} that yields the least value for the norm:

$$\|r_i - p_{ij}\|_w = \sqrt{\sum_{k=1}^{n} w_k ((\text{value}(q_{ik}) + \text{range}(q_{ik})) - (\text{value}(q'_{ijk}) + \text{range}(q'_{ijk})))^2}.$$

3.2.2 – Service Level Agreement

A SLA is of the form

$([t_1, t_2], \langle resource\ assignment\rangle_1, \langle resource\ assignment\rangle_2, \ldots, \langle resource\ assignment\rangle_r)$

where the range $[t_1, t_2]$ is the time interval over which the SLA is valid and a $\langle resource\ assignment\rangle$ is of the form $(type, value, w)$ where *type* is a member of the set *RTYP* of resource attributes, *value* is a QoS level expressed as an integer, and w is the importance level expressed as an integer.

RTYP contains the string names of various attributes of resource types under management and is partitioned according to these types. For example,

$RTYP = RTYP^{network} \cup RTYP^{disk} \cup RTYP^{CPU} \cup RTYP^{memory}$, where

$RTYP^{network}$ might be {bandwidth, packet-loss rate, jitter, delay}.

A resource assignment *<resource assignment>* specifies an agreement to provide the resource *type* at QoS level *value*. An assignment is said to evaluate to *true* if the actual QoS level equals or exceeds that in the assignment, and *false* otherwise. Examples of SLAs are: $SLA_1 = ([t_1, t_2],\ (memory,\ 24,\ 1))$

$SLA_2 = ([t_1, t_2],\ (memory,\ 24,\ 1),\ (bandwidth,\ 10,\ 2))$

A type in a resource assignment in a SLA must be measurable and quantifiable during service execution. A SLA has two important properties:

- A SLA is *atomic* – its resource assignments are sufficient to determine its status.
- A SLA is *satisfiable* – it evaluates to *true* under some interpretations. An evaluation to *false* during a service session indicates a SLA violation. Only assignments that are dynamically monitored can become *false* during service execution.

The reader will recall that a SLA is an agreement between two, or more, collaborating entities. In the simplest case, these are assumed to be a client and a service provider. Three key abstractions in the QoS model are a set '*SP*' of service providers, a set '*SER*' of services and a set '*RES*' of resources. These abstractions allow one to decouple a service provider from the resources it uses to execute a service – for example, certain resources may be owned by others, i.e. the SP may not own the resources. A service provider can offer one or more services, and must support a hosting environment, such as Apache Axis/Tomcat for WSs. A service may use one or more resources to execute.

The set *RES* of resources, similar to *RTYP*, is partitioned by the types under management. For example, $RES = RES^{network} \cup RES^{disk} \cup RES^{CPU} \cup RES^{memory}$ where, for example, $RES^{network} = RES^{bandwidth} \cup RES^{packet\text{-}loss\ rate} \cup RES^{jitter} \cup RES^{delay}$

Network attributes, such as bandwidth, packet-loss rate, jitter and delay are associated with $RES^{network}$. Similarly, attributes such as seek time, I/O throughput and storage capacity are associated with RES^{disk}. The set *RES* of available resources is a

union of resource sets based on their attributes. Each set $RES^{attribute}$ contains multiple instances of resources of the associated attribute.

Finally, each resource $R \in RES^{attribute}$ is associated with a single value; the amount required is denoted *val(R)*, specifying an appropriate measure of the resource instance, for example, a network bandwidth of 10 Mbps or a main memory of 512 MB.

To execute a service with a particular quality, as stated in a SLA, it is necessary to select resources (network, disk and processor) based on measurable attributes associated with the available services. Resource selection is driven by the fact that different resource instances provide different QoS levels.

Each resource assignment in a SLA corresponds to a resource request and importance weight pair in the service request. Each resource assignment in a SLA must provide a QoS level at least as great as its corresponding lower bound QoS level in the service request, but not greater than the upper bound specified by the range. This range element increases the flexibility for a client/application to request a range-based quality level; alternatively the value of the range must be zero if the client/application requests a fixed quality level.

To make this link between a service request $SREQ_i$ and its corresponding SLA_i, the sequence of resource assignments in SLA_i corresponding to the resource request sequence $r_i = (q_{i1}, q_{i2}, \ldots, q_{in})$ in $SREQ_i$ is denoted as $r'_i = (q'_{i1}, q'_{i2}, \ldots, q'_{in})$ where each q'_{ik} meets, or exceeds, the quality level specified in q_{ik} . That is, $type(q'_{ik}) = type(q_{ik})$, $value(q'_{ik}) \geq value(q_{ik})$, and $value(q'_{ik}) + range(q'_{ik}) \leq value(q_{ik}) + range(q_{ik})$.

3.2.3 – Service Level Agreement Formation

From the client's side, a SLA is a contract to receive a service with specified quality levels; from the QoS manager side, it is a commitment to deliver a service, based on resources with the specified QoS levels. The model described here attempts to capture these views in the abstract and does not address SLA protocols or reporting mechanisms such as those described in Chapter 2 (Bhoj *et al.* 1998; Pard *et al.* 2001; Nguyen *et al.* 2002).

The negotiation process involves a client initially proposing a SLA to the QoS Manager via a service request. The Manager replies with a yes/no type answer; if the reply is no, i.e. the Manager cannot satisfy the client/application request, the client/application should submit another request, perhaps with different QoS levels. Once a SLA is agreed between the two parties, the QoS Manager must reserve, and subsequently allocate, sufficient resources to meet the resource QoS levels in the SLA. This negotiation approach is based on a request/reply paradigm, and can be extended to support a 'counter-offer'; instead of replying with a 'no' answer, the manager could reply with a 'no', and a suggestion for possible resource reservation, similar to the approach taken in Hafid *et al.* (1998) as discussed in Chapter 2.

3.2.4 – Utilisation Model

Given a SLA_i, let $cost(q'_{ik})$ denote the cost of providing the resource specified in q'_{ik}. The evaluation of $cost(q'_{ik})$ can simply be based on a look-up table, or may be dynamically calculated as a service executes. As a service uses a collection of resources, the aggregate cost is the sum of the costs of the resources specified in SLA_i, namely:

$$Service\ cost\ for\ client\ i\ = \sum_{k=1}^{n} cost(q'_{ik})$$

Depending on whether the cost for executing a service is calculated by a client or a provider, one may optimise this cost from different perspectives. Given a particular QoS level, one may be interested in identifying a set of resources that can offer the QoS at a minimum cost. This would require a search to determine the best resource ensemble that offers a particular QoS at the minimum cost. To achieve this, it is necessary to keep the QoS level constant and search for a resource ensemble satisfying:

$$Min\_Cost_i = \min_{R \in RES^{type(q'_{ik})}} \int_{t1}^{t2} \sum_{k=1}^{n} cost(q'_{ik})_R \, dt$$

where $t1$ and $t2$ denote the validity time interval for SLA_i. Thus the Min_Cost_i equation assigns, for all SLA_i elements, resource $R \in RES^{type(q'ik)}$ to q'_{ik}, where all resource assignments satisfy SLA_i, with the cost of SLA_i thus minimised.

Alternatively, one may be interested in maximising the revenue that can be obtained by selecting from available resources while still satisfying the SLA. That is, to find a set of resources satisfying: $Max\_Cost_i = \max_{R \in RES^{type(q'_{ik})}} \int_{t1}^{t2} \sum_{k=1}^{n} cost(q'_{ik})_R \, dt$

The Max_Cost_i equation assigns, for all SLA_i elements, resource $R \in RES^{type(q'ik)}$ to q'_{ik}, where all resource assignments satisfy SLA_i, with the cost of SLA_i thus maximised.

One can compute the total cost to m clients, for a given set of SLAs, as:

$$Total\_Cost = \sum_{i=1}^{m} \int_{t1}^{t2} \sum_{k=1}^{n} cost(q'_{ik})\, dt$$

3.2.5 – Optimisation Problem

When considering QoS issues for a particular service provider, one may, given a sequence of SLAs, SLA_1, SLA_2, ..., SLA_m, allocate resources so that all SLAs are satisfied and total cost is minimised. The problem consists of two parts: evaluation of each SLA and cost optimisation. The evaluation of a SLA may be binary – i.e. true or false.

Recall that, given a SLA_i, $value(q'_{ik})$ denotes the number of $type(q'_{ik})$ QoS units specified in r'_i. The optimisation problem is to find an r'_i, such that, for each resource type $type(q'_{ik})$, the assignment does not exceed the resource capacity of the service provider, i.e. $value(q'_{ik}) \leq \sum_{R \in RES^{type(q'ik)}} val(R)$

and which maximises the total cost of all m SLAs:

$$Max_{TotalCost} = \max_{i} \sum_{i=1}^{m} \int_{t1}^{t2} \sum_{k=1}^{n} cost(q'_{ik})\, dt \ .$$

Thus the $Max_{TotalCost}$ equation maximises the total cost of all given SLAs without exceeding the service provider's resource capacity. This cost maximisation heuristic is consistent with the objective of maximising resource utility; thereby increasing revenues. This optimisation model focuses on the service provider side.

3.2.6 – Service Level Agreement Compliance

Having allocated resources for the specified QoS levels, it is important to ensure that, during an active QoS session, SLA compliance is maintained and all QoS attributes are satisfied. One approach is through a *monitoring service* that periodically reviews the status of the allocated resources, and an *adaptation service* that compares the agreed-on QoS levels with those actually provided. The monitoring service captures QoS values during the actual runtime and compares these to the values stated in the SLA. The adaptation service compensates for QoS degradation where possible, if such compensation is not possible, a violation report is made to the QoS Manager. A further discussion on adaptation can be found in Section 3.3.3.

3.3 Quality-of-Service Management

To realise some of the QoS management functions in this model, as described in Chapter 2, a mechanism for advance resource reservation is presented. This reservation mechanism is mainly intended to provide a degree of assurance that the reserved resources will be available for use; for increasing system flexibility and maximising resource utilisation.

3.3.1 – Advance Resource Reservation

A mechanism to reserve a collection of grid resources is proposed. It is important that the reservation be for a collection of resources, as most current grid applications require a collection of resources to run successfully. This co-reservation feature distinguishes the model from others such as GARA and VAS (Foster *et al.* 1999; Keahey and Motawi, 2003). The fundamental problem with advance reservation, (as discussed in Karsten *et al.* 1999), is that once an advance reservation is granted it is difficult to utilise or grant reservations during the *hold-back time* – the interval from the reservation being submitted until the start time. The problem arises when a client requests an immediate reservation for an indefinite period, which may overlap a previously granted advance reservation. A number of solutions have been proposed to solve this problem; for example, all reservations, including immediate reservation, could be specified within an interval – i.e. indefinite reservation is not supported.

Another solution is to partition resources for immediate reservation, and to only allow advance reservations for specified durations.

The solution adopted for the proposed model is for all reservations to be accompanied by duration specifications. This is a valid restriction for high performance (or high demand) resources, for applications such as scientific experiments or simulations. In these cases, there is prior knowledge of the need for resources and there are no *ad-hoc* requests for simple resources. Although this type of application (e.g. scientific experiments, simulations) would have prior knowledge of when the application needs to use resources, the resource configuration to deliver the desired QoS remains an issue. One approach to overcoming this issue is to utilise prediction systems, such as the *PACE* project (Jarvis *et al.* 2003), which would propose an estimate of the resource configuration required to deliver a certain QoS for a specific hardware platform – given that PACE has prior knowledge of the particular application.

An advance reservation model is specified in terms of five parameters:

t_s:	the reservation start time
t_e:	the reservation end time
cl:	a reservation class of service – *guaranteed, controlled load,* or *best effort*; discussed in Section 3.3.3.1.
$type \in RTYP$:	a resource type
value:	an integer specifying an attribute value for a resource of type *type*.

A *reservation request is* denoted as $Res(t_s, t_e, cl, (type_1, value_1), ..., (type_n, value_n))$ representing a co-reservation for n resources, with start time t_s, end time t_e, and reservation class *cl*. Each resource is specified by a type $type_k$ and an associated attribute value $value_k$. In the proposed QoS management model these reservation parameters result from the negotiation process with the client/application in establishing the SLA. A mechanism for *pre-emption priority* (Karsten *et al.* 1999) is assumed, to allow higher priority service executions to reduce the priority of services already running. The pre-emption priority ensures that when the reservation is not in effect, either before, or after, the reservation period, the job, or service, making use of the reserved resource is not refused or eliminated, but is rather assigned a low

priority value, which means switching its status from a *guaranteed* to a *best effort* type of service. To support pre-emption priority in practice, the underlying resource manager should be a priority-based system such as the Dynamic Soft Real-time (DSRT) scheduler (Chu and Nahrstedt, 1999).

3.3.2 – Admission Control

Admission control is the process of granting, or denying, reservation requests based on factors such as the actual load on a specified resource, and the policy that governs who, how and when reservation for a resource should be granted. The maximum available capacity for all resources of type *type* can be defined as

$$maxavail(type) = \sum_{R \in RES^{type}} val(R) \ .$$

The load on all resources of type *type* at time *t* can be defined as

$$load(type, t) = \sum_{R \in RES^{type} \wedge reserved(R,t)} val(R) \ .$$

where *reserved(R, t)* is true if resource *R* is reserved at time *t*.

The process of admission for a reservation request

$$Res(t_s, t_e, cl, (type_1, value_1), ..., (type_n, value_n))$$

can be formally described by a Boolean function that returns *true* if the request can be granted, and *false* otherwise, as in Algorithm 3.1.

Input: $Res(t_s, t_e, cl, (type_1, value_1), \ldots, (type_n, value_n))$
Output: *boolean*

```
1. for i = 1 to n
2.     for t = t_s to t_e
3.         if value_i > (maxavail(type) – load(type, t)) then
4.             return false
5.         end if
6.     end for
7. end for
8. return true
```

Algorithm 3.1: Admission Control Function

3.3.3 – QoS Adaptation

Adaptation is a key function of QoS management during the active phase of a session. Three scenarios under which adaptation can arise are:

- **Scenario 1 – New Service Request:** a service request is received for which there are insufficient resources. Adaptation can be used to free resources by adjusting the allocations of active services, for example, services whose clients indicate a willingness to accept a degraded QoS, such as receiving the lower boundary of the acceptable QoS in their SLA.
- **Scenario 2 – Service Termination:** a service completes successfully and its resources are released. Adaptation can be used to increase resource allocation for certain active services while still satisfying their SLAs. This can be realised by upgrading the quality level of services that have had their levels reduced, upgrading the levels of those not currently receiving the best quality specified in their SLAs. In other words, these services have valid SLAs, but the service quality being offered is at the lower boundary of the acceptable range.
- **Scenario 3 – QoS Degradation:** the situation where QoS falls below the minimum specified in a SLA. The degradation is detected, either by the resource monitoring service, or by an explicit notification from the underlying resource manager. Adaptation is used, if possible, to restore the degraded QoS to one satisfying the SLA.

The following sections describe a QoS adaptation scheme to address the scenarios described above. Section 3.3.3.1 describes the QoS classes supported by the scheme. Section 3.3.3.2 discusses the SLA and how it is used. Section 3.3.3.3 discusses the optimisation heuristic for adjusting resource allocation; to optimise resource utilisation. Section 3.3.3.4 presents the adaptation algorithm, based on reserving extra resources for *guaranteed* services, while general adaptation strategies are presented in Section 3.3.3.5. An example is presented in Section 3.4.

3.3.3.1 QoS Classes

Service delivery is categorised into three distinct classes of service motivated by the IETF: *guaranteed* (Shenker *et al.* 1997), *controlled load* (Wroclawski, 1997), and *best effort*. *Guaranteed* class service provides QoS based on pre-defined constraints

identified by the user, and agreed on by the provider. These constraints are specified using pre-agreed parameters, and must be supported by the service provider. QoS parameters are enforced to explicitly identified values and are monitored; the service provider is committed to delivering the service exactly as specified in the SLA. With *controlled load* service, users state their QoS requirements based on parameter ranges; a service provider must be able to offer QoS within the specified range. With *best effort* service, no SLA is required; i.e. there is no QoS agreement, and any suitable available resources are allocated to the client. This *best effort* service is the default situation on the Internet.

3.3.3.2 SLA and QoS Adaptation

Choosing an appropriate adaptation strategy and its constituent parameters relies on terms that are agreed on, in advance, during SLA establishment. Such terms involve, for example, acceptable levels of resource QoS, budget constraints and SLA violation penalties.

One important parameter, based on the selected class of service, is the level of acceptable QoS. For example, in the case of *controlled load*, a client/application specifies the range of acceptable QoS. This gives the QoS manager flexibility to support a range of acceptable quality levels for this particular client/application. With this flexibility the manager can upgrade, or downgrade, quality levels while still satisfying the SLA, aiming to maximise resource utilisation.

Such parameters in the SLA play a major role in constraining the adaptation strategy. They assist in better optimisation decisions as to which services should be upgraded, or downgraded, while maintaining SLA conformance and maximising resource utility.

3.3.3.3 Resource Allocation Optimisation

Many different clients can concurrently request service with a specific QoS requirement. The QoS levels must be negotiated, and agreed, along with other management parameters such as service name, class and duration. The optimisation heuristic, introduced above in Section 3.2.5, to reconcile these competing requests, is to maximise the total cost as defined by:

$$\max_{i} \sum_{i=1}^{m} \int_{t1}^{t2} \sum_{k=1}^{n} cost(q'_{ik})\, dt \ .$$

The QoS manager implements this heuristic by varying resource QoS levels based on the specified ranges in the SLA. This maximises overall resource utility, while maintaining acceptable quality for a user. This variation is undertaken for all active services, aimed at reaching the optimal resource allocation that satisfies the heuristic. The benefit of specifying QoS levels as ranges in the SLA provides flexibility for the QoS manager in allocating resources, and improves resource utilisation, by accommodating more service requests.

3.3.3.4 Adaptation Algorithm

Unlike the optimisation heuristic, the adaptation algorithm only operates on the *guaranteed* and *best effort* service classes. Under this approach, the system administrator determines the total *resource capacity* available for the *guaranteed* and *best effort* users, including processor, network and disk storage. In addition, an *adaptive capacity* can be specified, based on the rate of resource failure, or congestion, as determined by the system administrator. The adaptive capacity is used when the QoS for the *guaranteed* clients has been degraded; as a means to compensate for such QoS degradation, or to be used by *best effort* users when it is not needed for *guaranteed* clients. The algorithm incorporates a *minimum capacity* for *best effort* clients, also determined by the system administrator. Providing a minimum capacity for *best effort* clients is useful in distributed systems and shared-resource environments, because services with no SLAs – i.e. without QoS guarantees – will not be starved of resources, as they are likely to receive a low level of resource usage. The concept of adaptive capacity is an extra resource ability, to be used when adaptation is needed in terms of Algorithm 3.2.

These capacity allocations are dynamic, in that, using the adaptive and *guaranteed* capacities, the *best effort* capacity utilises the free adaptive resources, provided they are not currently allocated. The algorithm starts execution by invoking either the *Allocate_Guaranteed_Resource* or the *Allocate_Best_Effort_Resource* function, as shown in Algorithm 3.2:

- ***Allocate_Guaranteed_Resource*:** when a request arrives, at line 23 in Algorithm 3.2, to allocate resources for *guaranteed* clients, a check is made to find out if the request is less than stated in the SLA; if the request is for less, it is considered, while, if the request is for more then only the specification in the SLA is considered; (lines 26 and 28). At line 26, if there are insufficient resources for allocation to the specification in the SLA, then the adaptation function *Adapt* is called. The *Adapt* function, at line 17, calculates the available net capacity for *guaranteed* clients at that time. If there are insufficient resources available at *guaranteed* capacity, then it borrows resources to satisfy the SLA under consideration, from the adaptive resource capacity, and make the remainder of the adaptive resource capacity available for *best effort* clients.
- ***Allocate_Best_Effort_Resource*:** when a request is made for allocation of resources to *best effort* clients, (at line 33), the algorithm calculates the net capacity for the *best effort* clients at that time; which is the sum of the pre-defined *best effort* resource capacity and the available adaptive capacity, i.e. the unused capacity of the adaptive. If the calculated net capacity is insufficient for the request under consideration, a rejection message is generated, otherwise the request is honoured.

Figure 3.3 illustrates the dynamic property of the adaptation algorithm.

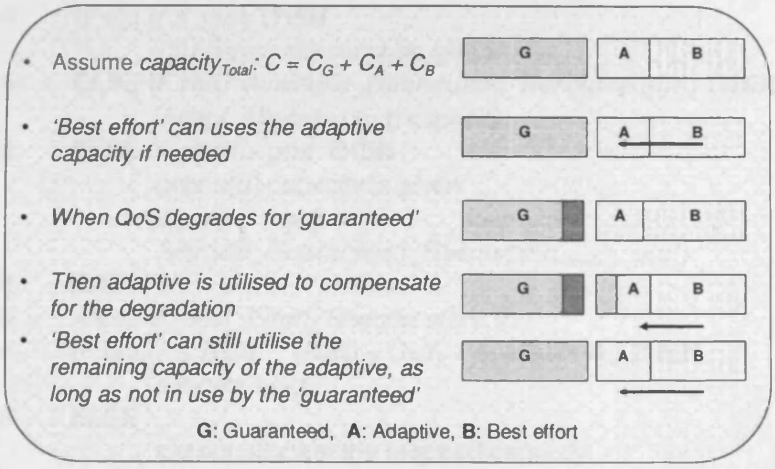

Figure 3.3: The Dynamics of the Adaptive Algorithm

1 C: the total resource capacity
2 C_G: the *guaranteed* QoS capacity
3 C_A: the adaptive capacity
4 C_B: the *best effort* QoS capacity
5 N_G: net capacity for *guaranteed*

6 **Then** $C = C_G + C_A + C_B$

7 U: set of ALL clients $U = \{u_1, ..., u_n\}$
8 G: set of users of class *guaranteed* $G = \{v_1, ..., v_n\}$
9 B: set of users of class *best effort* $B = \{w_1, ..., w_n\}$

10 $c(u,t)$ = capacity required at time t by client $u \in G$
11 $b(u,t)$ = capacity required at time t by client $u \in B$
12 $g(u)$ = *guaranteed* capacity with a SLA for client $u \in G$

13 *Available_Guaranteed_Resource (g(u))*
14 **IF** $\sum_u g(u) \leq C_G$; *where* $u \in G$ **THEN**
SLA guarantees to $g(u)$ can be honoured
16 **ENDIF**

17 *Adapt()*
18 Net capacity $N_G(t) = C_G(t) - \sum_u g(u)$; where $u \in G$
19 **IF** $N_G(t) < 0$, (guarantees cannot be honoured at time t) **THEN**
ADD $(\sum_u g(u) - C_G(t))$ from A to G
ADD $(C_A(t) - [\sum_u g(u) - C_G(t)])$ from A to B
22 **ENDIF**

23 *Allocate_Guaranteed_Resource(c(u,t), g(u))*
24 **IF** c(u,t) ≤ g(u) **THEN**
$c(u,t)$ capacity must be given
26 **ELSE IF** NOT *Available_Guaranteed_Resource(g(u))* **THEN**
Adapt; allocate $c(u,t)$ capacity
28 **ELSE IF** $c(u,t) > g(u)$ **THEN**
only $g(u)$ capacity is given
$c_{new(u,t)}$ $g(u)$
Allocate_Guaranteed_Resource($c_{new(u,t)}$, $g(u)$)
32 **ENDIF**
33 *Allocate_Best_Effort_Resource(b(u,t))*
34 **IF** $b(u,t) \leq N_B(t)$; $(N_B(t) = C_B(t) + available\ \boldsymbol{A})$ **THEN**
allocate $b(u,t)$
36 **ELSE**
cannot allocate the required capacity
38 **ENDIF**

Algorithm 3.2: QoS Adaptation

The adaptation algorithm has two important advantages.

- Resources are never under-utilised; the extra capacity is used by *best effort* clients provided the capacity is not needed by *guaranteed* clients (Algorithm 3.2, lines 21 and 34).
- A minimum resource capacity is allocated for *best effort* clients (Algorithm 3.2, lines 4 and 6).

3.3.3.5 Adaptation Strategies of Grid Services

The adaptation scheme is based on Algorithm 3.2 and the resource allocation optimisation heuristic described in Sections 3.3.3.3. The QoS Manager periodically applies the optimisation heuristic, and if there is a considerable gain in benefit to the service provider, the resource allocation is modified. On receipt of a request from a *guaranteed* client, the adaptation algorithm is applied; if the request cannot be accommodated, the optimisation heuristic is executed.

3.4 Example

An example illustrates the operation of the adaptation scheme, with an emphasis on processor resources. Assume that a scientist is about to conduct a simulation experiment using grid services and infrastructure. The experiment is to run at site *A* on an SGI multiprocessor machine with *64 processors* and *10 GB* of memory. The database, holding the required data for the simulation, resides at site *B*. A second scientist participating in the simulation is located at site *C*. The resources required for the experiment are:

- A *622 Mbps* communication link to connect site *B* and site *A*.
- A *45 Mbps* communication link to connect site *C* and site *A*.
- *10 processor* nodes, *2 GB* of *memory* and *15 GB* of *disk space* at site *A*.

The resources must be available over the duration of the experiment – t_5 to t_9. The *SGI* machine is configured to provide *26 processor* nodes to all grid users, with the rest dedicated for local processing. The grid system operator partitions the *26 processor* nodes as:

$C_G = 15$, $C_B = 6$ and $C_A = 5$ *processor* nodes

$C = C_G + C_B + C_A = 15 + 6 + 5 = 26$ *processor* nodes

Three SLAs are negotiated with the QoS manager over the period t_5 to t_9:

- SLA_1: network *bandwidth* of *622 Mbps* from Site *B* to Site *A*. Using the SLA format outlined in Section 3.2.2, SLA_1 can be expressed as:

  $SLA_1 = ([t_5, t_9], (\ bandwidth,\ 622,(source:\ B,\ destination:\ A),\ 0))$

- SLA_2: network *bandwidth* of *45 Mbps* from Site *C* to Site *A*. Using the SLA format outlined in Section 3.2.2, SLA_2 can be expressed as:

  $SLA_2 = ([t_5, t_9], (\ bandwidth,\ 45,(source:\ C,\ destination:\ A),\ 0))$

- SLA_3: *10 processor* nodes, *2 GB* of *memory* and *15 GB* of *disk* space on the *SGI* machine at Site *A*. Using the SLA format outlined in Section 3.2.2, SLA_3 can be expressed as:

  $SLA_3 = ([t_5, t_9], (CPU,\ 10,\ 0),\ (\ memory,\ 2,\ 0),\ (disk,\ 15,0))$

Figure 3.4 depicts the three sites and resources required as in SLA_1, SLA_2 and SLA_3.

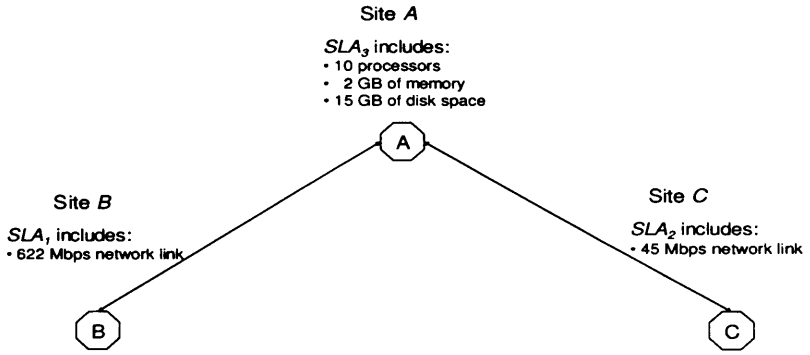

Figure 3.4: Sites and Established SLAs

Assume the following measurements are recorded during the period t_0 through t_9. The *'a'* and *'u'* notations correspond respectively to *available* and *used* processor node resources.

Notation:
- C_G : the *guaranteed* QoS capacity
- C_B : the *best effort* QoS capacity
- C_A : the adaptive capacity
- a : the number of available processor nodes
- u : the number of used processor nodes

- At $(t_0$ to $t_3)$ the processor node allocation is:

  C_G: $u = 10$, $a = 5$; processor node utilisation and availabity at C_G

  C_B: $u = 6$, $a = 0$; processor node utilisation and availabity at C_B

  C_A: $u = 0$, $a = 5$; adaptive capacity from C_G point of view with the corresponding processor node utilisation and availability.

  C_A: $u = 4$, $a = 1$; adaptive capacity from C_B point of view with the corresponding processor node utilisation and availability.

- At t_4:

  C_G: $u = 4$, $a = 11$; processor node utilisation and availabity at C_G

  C_B: $u = 6$, $a = 0$; processor node utilisation and availabity at C_B

  C_A: $u = 0$, $a = 5$; C_G point of view

  C_A: $u = 3$, $a = 2$; C_B point of view

  (*best effort* clients use resources in unpredictable patterns)

- At t_5: three processors from the C_G resource pool become inaccessible; C_G is therefore updated to *12 processor* nodes. SLA_3 is also due to be active; requiring the allocation of *10 processors*

  C_G: $u = 14$, $a = 1$; to be brought from C_A when required.

  C_B: $u = 6$, $a = 0$

  C_A: $u = 2$, $a = 3$; C_G point of view

  C_A: $u = 3$, $a = 0$; C_B point of view

- At t_8: three additional processors become accessible:

  C_G: $u = 14$, $a = 1$

  C_B: $u = 6$, $a = 0$

  C_A: $u = 0$, $a = 5$; C_G point of view

  C_A: $u = 3$, $a = 2$; C_B point of view

- At t_9: SLA_3 has completed its validity period:

  C_G: $u = 4$, $a = 11$

  C_B: $u = 6$, $a = 0$

  C_A: $u = 0$, $a = 5$; C_G point of view

  C_A: $u = 3$, $a = 2$; C_B point of view

This example illustrates how the adaptation strategy reserves resource capacity for *guaranteed* clients; for use when there is a resource failure, or congestion. The dynamic nature of the strategy allows unused resources to be utilised by *best effort*

clients. *Best effort* clients can therefore always make use of system resources. This adaptation strategy is, furthermore, a generic approach and is not restricted to a specific type of resource, unlike other work (Oguz *et al.* 1998; Foster *et al.* 2000). Adaptation strategies based on reserving *'just sufficient'* resources (Chu and Nahrstedt, 1999; Cardei *et al.* 2000) are not used here, as it is difficult to apply such mechanisms to different types of resources.

3.5 Summary

A new model for resource management based on QoS is presented. The model shows that the QoS problem – to determine, given multiple client requests, the optimal resource allocation that maximises utilisation and maintains requested QoS levels – is an optimisation problem. A heuristic to achieve this is described. The model is SLA-based, with a client negotiating for service access during an establishment phase. The model selects services based on their QoS properties, as published by a service provider. Selecting services based on QoS properties requires a registry service that can recognise services with such QoS properties, such as the extended version of UDDI (ShaikhAli *et al.* 2003).

The model employs a new mechanism for advance resource reservation, able to reserve one or more resources. A novel approach for QoS adaptation, to compensate for resource shortages when resource QoS degrades, is introduced in this model. Finally, an example illustrating the adaptation approach is given.

Chapter 4 ~ Framework Design presents the design for the G-QoSm system, based on the model presented in this Chapter.

Chapter 4 ~ Framework Design

4.0 Background

Some applications utilising a grid computing infrastructure require the simultaneous allocation of resources, such as computer nodes, network bandwidth, disk storage or other specialised resources. Collaborative work, visualisation and image processing in distributed computing are examples of such applications. As such applications operate in a collaborative mode, data must be stored and delivered in a timely manner to clients or processing nodes, and sufficient processing power must be available to process the data according to the required behaviour; consequently such applications have QoS requirements.

4.1 Synopsis

This Chapter presents a novel architecture for QoS management, called G-QoSm; based on the conceptual model described in Chapter 3.

G-QoSm is a general-purpose architecture, in the sense that it can be applied within various SOAs, such as computational grids. It has a number of features:

- A negotiation protocol, between a client and QoS management entity, or QoS Manager, is used interchangeably on behalf of a *service provider*. This negotiation process either results in an agreement, i.e. the establishment of a SLA, or finds no agreement. If in agreement, the SLA constitutes a contract whose elements i.e. values associated with QoS properties, must be supported throughout the agreed-on QoS session.
- A registry structured to allow a service provider to publish its services with QoS properties, hereinafter referred to as a QoS-aware registry. This allows services to be found based on QoS properties. The discovery process employs search mechanisms for searches based on complex discovery requests, constructed using operators, such as '=', '<', '≤', '>', and '≥', and the logical operators *AND* and *OR*.

- A mechanism for selecting a service based on its QoS properties. Different resource allocation strategies for computing QoS are used: *resource-domain* for relatively small applications and services, and *time-domain* for applications and services requiring high-performance resources.
- A design with a resource reservation module decoupled from the underlying Resource Manager (RM). This decoupling adds flexibility, in that new types of RMs can be incorporated as they become available. This flexibility is made possible through an intermediate software interface, which integrates a newly-introduced RM with the existing reservation module.

4.2 Framework Overview

G-QoSm is intended to operate in a SOA, and the basic principles of SOAs, as in Figure 4.1 (including ***publish, find*** and ***bind***) (Graham *et al.* 2002), should hold.

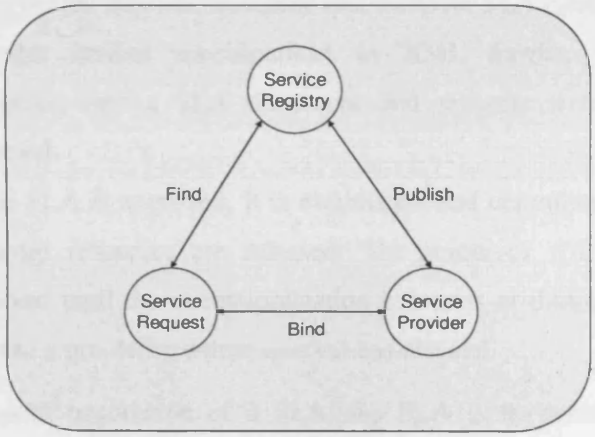

Figure 4.1: Concept of a Service-oriented Architecture

A major contribution in this project is an enhancement of the basic principles of SOAs with resource QoS provisions, allowing *publishing* of services with QoS properties, *finding* services based on QoS properties, and *binding* to services with resource QoS provisions.

At a conceptual level, G-QoSm operates as follows:

- Service providers publish their services to the QoS-aware registry with QoS properties for each service. These properties can be qualitative, such as reliability and accuracy, or quantitative for resource characteristics such as network bandwidth. The service properties are stored in a service profile for later use. For the purpose of this thesis, quantitative characteristics are considered foremost.
- A client submits a service request, with optional QoS properties, to the QoS Manager, which takes clients' requests on a 'First in First out' basis (FIFO).
- The QoS Manager attempts to find a suitable service, based on the specifications supplied by the client. Where no specifications are supplied by the client, the QoS Manager relies on the service profile created during the service-publishing process. In all cases, the QoS Manager:
  - Queries the QoS-aware registry for possible matching services.
  - Selects the most suitable service.
  - Reserves the required resources and waits for SLA establishment.
  - Encodes service specifications in XML format, noting reserved resources, into a SLA document and presents it to the client for approval.
  - If the SLA is approved, it is established and committed; otherwise the reserved resources are released. The resources will be temporarily reserved until the client/application approves or disapproves the SLA, or until a pre-defined time interval has elapsed.
- On successful negotiation of a SLA, the SLA is forwarded to the client, together with its SLA-identifier (SLA-ID), for a later service activation request.
- When the SLA validity period approaches, the client can request the service, with the QoS specified in the SLA, and the service is then made available for the full SLA validity period, with a start and end time to define its validity period.

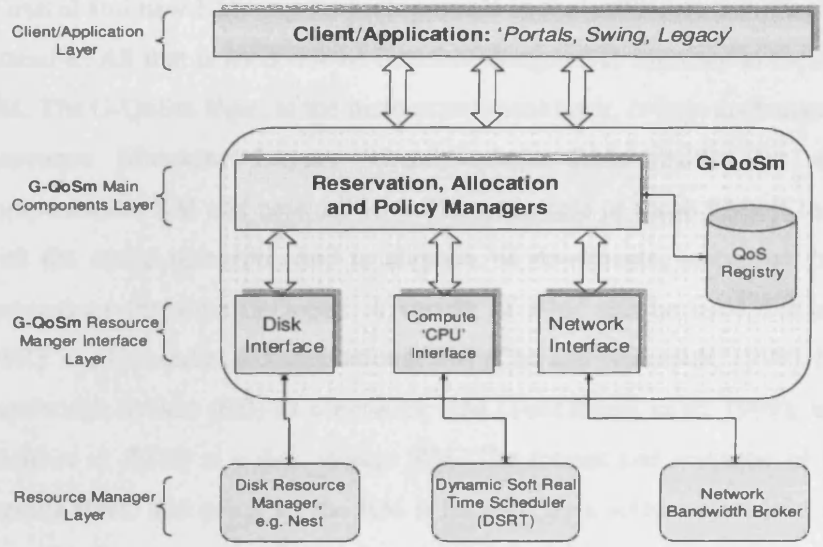

Figure 4.2: The G-QoSm Framework: A Conceptual View

The G-QoSm Framework in Figure 4.2 shows a 4-layer architecture, including a:

- **Client Application Layer:** where the client/application resides and interacts with the G-QoSm framework. The client/application can access the framework via various means, such as portals, swing libraries and legacy applications. This interaction with the framework is possible through the G-QoSm API, where the client/application can interact with the G-QoSm framework, can request services and negotiate SLAs.
- **G-QoSm Main Components Layer:** where reservation, allocation and policy managers G-QoSm components are found – further details on each component are given in Section 4.6. These components interact with the client/application for service requests and SLA negotiations. They also interact with the various resource manager interfaces to allocate and de-allocate resources. These components are designed to interact with various RMs in a uniform way. They are not designed for a particular RM, and a RM interface layer is therefore needed to interact with specific RMs.
- **G-QoSm Resource Manager Interface Layer:** where interfaces for various RMs exist. These interfaces are designed to translate instruction from the G-QoSm main component layer to the underlying RM layer. This translation essentially converts instructions from the G-QoSm main component layer to instructions which can be understood by designated RMs. This interface layer

is useful and new RMs can be incorporated in the architecture as they become available. All that is needed is an interface designed to translate to the specified RM. The G-QoSm logic, at the main component layer, is kept unchanged.

- **Resource Manager Layer:** where various RMs reside, for example, computational RM and network RM. The main role of these RMs is to interact with the actual resources, and to allocate, or de-allocate, resources, based on instructions from the G-QoSm. A variety of RMs can be used, for example, DSRT can be used as a computational RM (Chu and Nahrstedt, 1999), Network Bandwidth Broker (BB) as a network RM (Teitelbaum et al. 1999), and Nest (Bent et al. 2002) as a disk storage RM. The format and semantic of the data coming from, and going to, the RM is handled by a software module, called a *wrapper*, designed specifically for each type of RM. A further discussion on RM integration can be found in Section 5.2.3.

This layered architecture is flexible, and can be realised by an ability to incorporate new RMs as they become available, which only involves designing a specific software wrapper for the RM introduced, while the main components of the G-QoSm design are not affected.

4.3 G-QoSm Architecture

G-QoSm has three main operational phases, as described in Chapter 2; ***establishment***, ***activity*** and ***termination***. During the establishment phase, a client application specifies a desired service and the QoS requirements. G-QoSm then undertakes a *service discovery*, based on the specified QoS properties. This process submits a *service request* query to the QoS registry, and receives a list of matched services available. G-QoSm then selects a suitable service and presents an agreement offer 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 (following a resource reservation expiry, an agreement violation or service completion); resources are then freed for use by other clients. G-QoSm supports these three phases using specialist components, as depicted in Figure 4.3. Subsequent sections describe these interactions and highlight how service provision occurs.

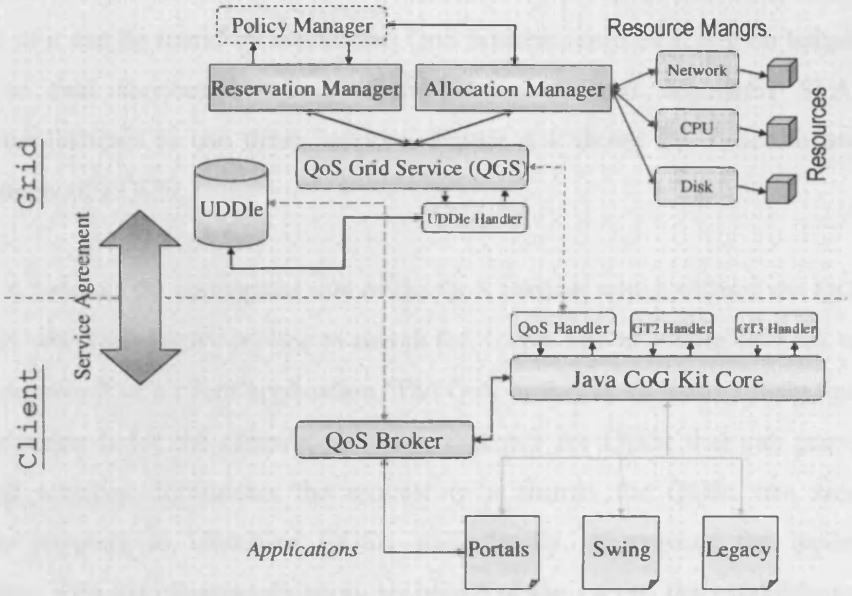

Figure 4.3: G-QoSm Architecture

The Client/Application accesses the QGS through the Java CoG Kit and QoS Broker

4.4 QoS Grid Service

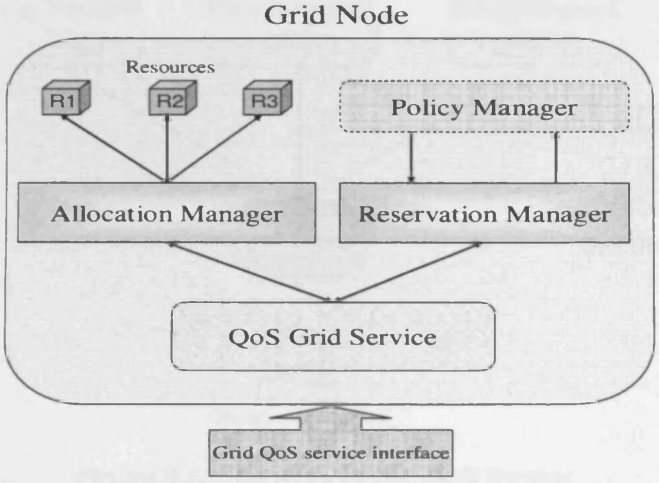

Figure 4.4: Structure of a QGS

The basic component of the G-QoSm architecture is the QoS Grid Service (QGS), an OGSA-based grid service, providing QoS functionality, including negotiation, reservation and resource allocation, accessed through its service-interface operations.

Each QoS-enabled resource is accessed via a QGS, which publishes itself to a registry service so it can be found by clients and QoS brokers (entities acting on behalf of the client to find services based on QoS properties), and negotiates SLAs with clients/applications to use these services. Figure 4.4 shows the structure and main components of a QGS.

Figure 4.5 shows the conceptual role of the QoS Broker, which utilises the QGSs and interacts with the registry service to search for, locate and negotiate services with the QGSs on behalf of a client application. The QoS broker is an intermediate agent that accepts requests for the client/application, searches for QGSs that can provide the required services, formulates the request in a format the QGSs can recognise, submits requests to identified QGSs, and, finally, aggregates the replies and negotiates with the client/application, on behalf of the QGSs, the establishment of a SLA. This process simplifies the client/application role, especially when dealing with multiple grid nodes, involving coordination of multiple requests, negotiation with multiple QGSs and aggregating the SLAs.

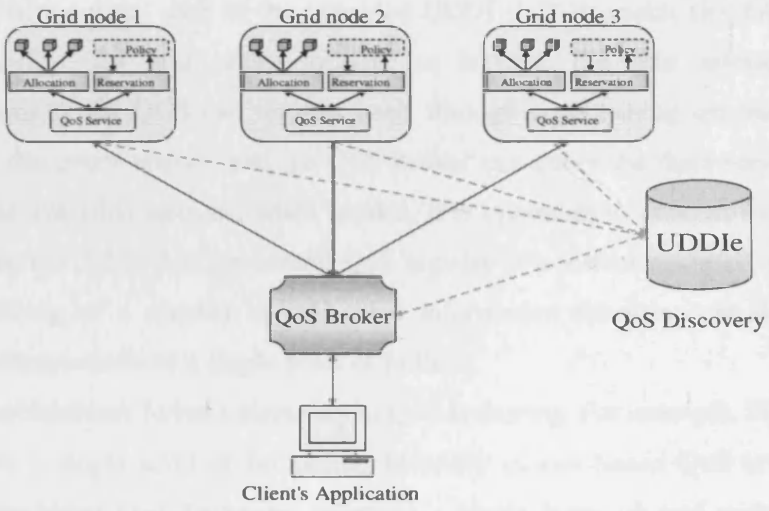

Figure 4.5: The Role of the QoS Broker

The following Section outlines the benefits of the basic QGS building block, of the G-QoSm architecture, when used by a QoS Broker.

4.5 QoS Brokering

The concept of the QGS, together with the concepts of a QoS Registry and QoS Broker, incorporates various features:

- It hides, from client/applications, information about locations and specifications of each QoS-enabled grid node, and delegates this task to the QoS Broker.
- It simplifies the task of a client/application when requesting multiple grid nodes – the client needs only state, for example, the number and specifications of the QoS-enabled grid resource, and the QoS broker locates the specified resources, if available.
- The QoS broker-based approach provides scalability; for example; when a grid node joins or leaves the grid its state information is maintained in the QoS Registry and not in the QoS Broker, improving scalability and flexibility. This is possible because detailed information for this QoS-enabled grid node is retained in the QGS, which is, in essence, a grid service representing a physical grid node. Publishing the service (e.g. QGS) in a QoS discovery system such as the extended UDDI (UDDIe) adds flexibility, and scalability, for grid nodes joining, or leaving, the grid infrastructure. Essentially the QGS can register itself through a publishing process to the QoS discovery system, and the QoS Broker can query the discovery system on the available services, when needed. It is important to note, as mentioned in Section 2.2.2, that the central QoS registry is a virtual resource, possibly consisting of a number of replicated information services, and does not, therefore, constitute a single point of failure.
- The architecture forms a hierarchy of QoS brokering. For example, Figure 4.5 shows a single level of brokering, basically *cluster-based* QoS brokering. Cluster-based QoS brokering refers to a single layer of grid nodes – the Cluster Broker – which interacts and directly controls a group of grid nodes. However, this can be extended by introducing another level of brokering, called *grid-level* brokering as in Figure 4.6 – a two level brokering. This Grid Broker interacts directly with cluster brokers and not with the grid nodes; useful when simultaneously dealing with large numbers of grid nodes. A drawback of such QoS brokering is that as the depth of the hierarchy

increases, additional design complexity is introduced in the root broker. The root broker undertake the management process; sending requests to multiple brokering entities, and then aggregating their replies to ascertain if the original client request can be fulfilled, and a SLA can be established.

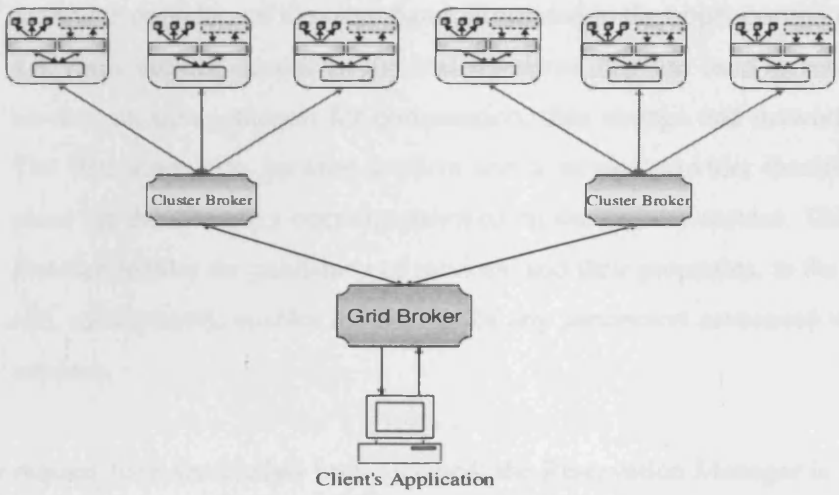

Figure 4.6: Hierarchical QoS Brokering

4.6 Components

The QGS interacts with various modules to deliver QoS guarantees. In addition to the main QoS functions, it supports two types of resource allocation strategies, allowing the client application to specify the strategy that best suits its needs. These strategies are:

- **Resource Domain:** A client can specify a certain percentage capacity for a shared QoS-enabled resource – for example, access to 50 % of processor time, or request for 20 Mbps bandwidth from 155 Mbps available.
- **Time Domain:** A client can request an entire resource for exclusive use – i.e. no other clients are allowed to share the resource. This functionality is enabled by ensuring all requests for resources are issued through the QGS.

The components of the QGS are the *Reservation Manager*, *Allocation Manager*, *QoS Registry Service* and *Policy Manger* as shown in Figures 4.3 and 4.4. The

architecture in Figure 4.3 consists of a client (the lower part of the figure), and a service provider (the upper part of the figure) in a grid environment.

- The client makes use of a registry service, (the UDDIe), to find services requested. A client may be a physical user accessing G-QoSm services, or may be an application.
- A service provider, on the other hand, illustrated in the upper portion of Figure 4.3, must provide access to physical resources that are used to manage the service, including support for computation, data storage and network access. The first interaction between a client and a service provider therefore takes place via the discovery operation invoked on the registry service. The UDDIe Handler enables the publishing of services, and their properties, to the registry, and, subsequently, enables the altering of any parameters associated with such services.

Once a request for a service has been received, the Reservation Manager is invoked, and, subsequently, the Allocation Manger undertakes resource allocation. To support QoS characteristics, a service provider must ensure that in addition to the service being offered to external users, it supports additional components to allow reservation, and subsequent allocation, of resources where the service is to be hosted. In addition, the service must be annotated with additional properties that enable these QoS attributes to be encoded in its interface.

The QGS undertakes resource reservation and 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 via the Reservation Manager, using the admission control function outlined in Algorithm 3.1, and, if such a reservation is possible, the requested resources are reserved, the reservation table (where reservation entries are stored) is updated, and an agreement, based on reservation specification, is generated and returned to the client.

When a resource allocation request is received (as in the case of computational QoS) the QGS undertakes a validation process, and verifies that the user has, indeed, made a reservation based on the supplied agreement. This test basically retrieves the reservation parameters from the reservation table and compares these with those

supplied by the client/application. If this test is passed, the QGS submits the specification of the job to be executed to the Globus Resource Allocation Manager (GRAM) for that particular resource. Along with the job specification, the QGS supplies other parameters related to computing resource allocation and QoS levels; these parameters are passed from GRAM to the computing RM for immediate allocation, as GRAM has a direct interaction with the compute RM, as described in Chapter 5, Section 5.2.4. This process is handled by the Allocation Manager in the QGS. For network QoS, when the active phase of the QoS session has started, the networking elements (e.g. a Bandwidth Broker) are configured to support the network QoS as specified in the SLA. Further details on QoS support are presented in Chapter 6.

4.6.1 – Reservation Manager

The Reservation Manager uses a data structure that supports reservations of 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, in turn, is responsible for validating reservation requests by applying domain-specific rules, established by the resource owners, as to when, how and by whom the resource can be used. The Policy rules are assumed as being supplied by the system administrator. In brief, when the Reservation Manager receives a reservation request from the QGS, it contacts the Policy Manager for validation, and then performs an admission control to check the availability of the requested resource. If successful, it returns a positive reply to the QGS, which allows the QGS to propose an agreement offer.

4.6.2 – Allocation Manager

The Allocation Manager primarily interacts with underlying resource managers for resource allocation and de-allocation, and to enquire about the status of resources. It has interfaces with various resource managers such as DSRT (Chu and Nahrstedt, 1999) and the Network BB_{Basic} (Sohail *et al.* 2003). When the Allocation Manager receives a resource allocation request from the QGS, it forwards the request to the designated underlying RM, through its specific interface, as outlined in Figure 4.2.

4.6.3 – QoS Registry Service

G-QoSm is intended for use within a SOA, and its implementation is based on an Open Grid Service Infrastructure (OGSI) (Foster *et al.* 2002). Essentially the core component of the G-QoSm, the QGS is a grid service. The QGS, and other grid services in the OGSI container, should be published to a registry service. However, service publishing here does not mean only publishing a service name, URL and basic description. A QGS includes information on QoS-enabled services it offers, what allocation strategies it employs, in the case of computing QoS provisions, and what classes of network QoS it offers. Such services, with their QoS information, are published in a *QoS Registry Service* so the service can be found, based on the QoS information. The QoS Registry Service is used, in this context, to publish services with their QoS properties.

4.6.4 – QoS Policy Manager

The Policy Manager aims to provide information about the resource characteristics, and rules governing when, what and who is authorised to use resources. This Policy Manager relies heavily on the existence of a policy repository – data storage for policies. Resource owners include information and rules, about their resources, in the policy repository; for example, resource capacity allowed for utilisation and class of service their resource can provide. These rules are utilised by the Policy Manager to provide information on resource characteristics and usage policies when resources are requested for reservation, and are mainly used for validating requests.

4.7 Java CoG Kit Core

4.7.1 – Background

The QGS manages grid resources that are QoS-aware. However, to take advantage of, and utilise, such QoS-aware grid resources it is important for applications to conveniently interact with such entities, without having to undergo significant changes. Consequently, interaction with the QGS is supported via middleware libraries, as a means to interact with the G-QoSm architecture.

The Java CoG Kit (von Laszewski *et al.* 2001) is Java-based middleware used to access various grid implementations, such as Globus Toolkit Version 2 (GT2) and Version 3 (GT3). One of the modules of the Java CoG Kit, called *cog-core* (Amin *et al.* 2004) provides the core functionality for technology and architecture-independent interoperability. *Cog-core* provides APIs offering abstract grid functionality such as remote job execution and file transfers without consideration of the underlying grid implementation. For example, consider a grid application developed using the APIs provided by *cog-core*. As *cog-core* offers abstract functionality, irrespective of the back-end architecture, whether GT2 or GT3, the same application can be executed on a variety of platforms. Thus, to run an application on a GT2 service, the user merely needs to state a provider attribute as GT2. The same application can later be executed on a GT3 service without modification to its implementation, by simply changing the provider attribute from GT2 to GT3.

Cog-core has the required functionality for mapping abstract application requirements into back-end specific detail, such as GT2 and GT3 detail, controlled by the corresponding provider attribute. To provide seamless interaction between grid applications and the QoS-aware grid resources, the functionality of *cog-core* is augmented by incorporating QoS-related parameters. The necessary logic and implementation overhead for QoS management is introduced into *cog-core*, thereby allowing an application to make use of QoS features by changing the provider attribute to QoS. The provider attribute, is an attribute the client application should specify to enable *cog-core* to select which back-end service to access, whether GT2, GT3 or QoS service.

4.7.2 – Constructs

The two basic constructs of the *cog-core* library, and enhancement to the QoS domain, are *Task* and *Handler*:

4.7.2.1 Task

A *task* in *cog-core* denotes an atomic unit of execution, abstracting remote job execution or a file transfer request. A task has a unique identity, a security context, a specification, a service contact and a provider attribute. The task identity helps

uniquely represent the task across the grid. The security context represents the abstract security credentials of the task, requested by the client who initiated the task. Most back-end grid implementations will have their own notion of a security context; the security context in *cog-core* offers a common construct that can be extended by an implementation to satisfy a back-end requirement. The specification represents the actual attributes required for the execution of the grid task. The generalised specification can be extended for common grid tasks such as remote job execution and file transfer request. The service contact associated with a task symbolises the grid resource required to execute it, and the provider attribute specifies the desired back-end grid implementation for the task.

4.7.2.2 Handlers

The task handler provides a simple interface to support interaction with a generic grid task. It categorises a submitted task, depending on the selected back-end service, and provides the appropriate functionality based on its provider attributes. *Cog-core* contains a separate handler for the back-end functionality it supports. These handlers map the generic grid parameters of a task into the back-end implementation-specific grid functionality. To incorporate the *cog-core* functionality into the QoS domain, a *QoS Handler* that holds the QoS-related implementation and logic is provided. The QoS Handler manages negotiation, task execution and data redirection between the client application and the QoS-aware grid resource. It is important to remember that a QoS-aware grid resource is the actual physical grid resource, while QGS is the grid service representing the grid resource, with the interaction between the QoS Handler and the QoS-aware grid resource achieved through the QGS.

To enable a grid application to request a network or computational resource with QoS provisions, certain configuration parameters are needed. The application developer must specify the QoS parameters to be considered during the negotiation, including start and end times, resource type and specifications. Once the task object has been specified, the QoS Handler is delegated, on behalf of the client, to negotiate QoS requests. In this case, the QoS Handler is seen as the client by the QGS. This is useful especially when an application requires more than one grid resource. All the application needs do is 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, and return an agreement if the negotiation succeeds.

4.7.2.3 Integration

Figure 4.7 depicts the architecture of the Java CoG Kit with the integration of G-QoSm's QoS Handler. This figure shows the modular design of a three-layered architecture: i) the client application layer, ii) the Java CoG Kit layer, and iii) the back-end services layer, whether GT2, GT3, WSRF, QoS or similar. The QoS services supported by G-QoSm only interacts with the QoS Handler, a module of the Java CoG Kit. Details of the logic needed to handle the communication with the G-QoSm are hidden from the client application, and are handled by the QoS Handler as part of the Java CoG Kit. It is important to note that the API used to access back-end services are similar, which makes it convenient to switch between back-end services, such as accessing GT2 or QoS services. In Figure 4.7 the *Reservation* and *Execution* Modules are designed in two parts – the *client* and *server*. The client section is part of the Java CoG Kit, namely the QoS Handler, and its role is to implement the logic needed for communication with the QGS, i.e. from the application perspective. The server part implements the interaction handling between the client and the services supported by G-QoSm, i.e. from the G-QoSm perspective.

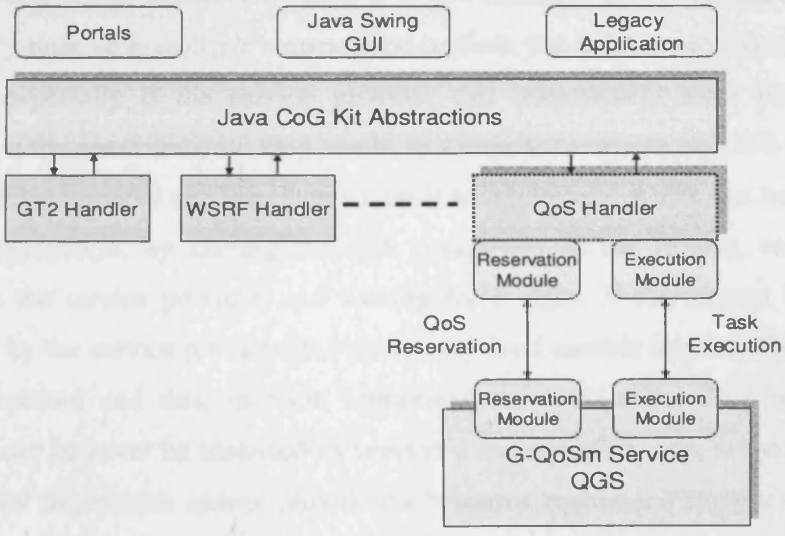

Figure 4.7: QoS Handler Integration with the Java CoG Kit

This abstraction of the Java CoG provides several advantages:

- The Java CoG Kit provides access to various grid implementations, through its API.
- The QoS service can merely be a back-end service, and focus on back-end functionality, while allowing the client application interface to be handled by the Java CoG Kit.
- Because the Java CoG Kit already has many grid applications using its API, these grid applications can easily utilise the QoS back-end service. This is particularly true because, with a minor change, an application already accessing grid services through the Java CoG Kit is able to use the QoS back-end services. For new applications it is a simple process to use the Java CoG Kit API, with more detail on the implementation is given in Chapter 5.

4.8 Negotiation of QoS Levels

A QoS negotiation is based on a request/reply paradigm, which can be as simple as a single request and reply, or can involve multiple requests and replies. The negotiation process must reach agreement, between the client and the service provider, about the reservation schedule, or the parameters involved in providing a given service, before the service starts. It is not necessary for a negotiation to take place every time, (e.g. multiple requests and replies), but at least one request/reply is required, especially if the service provider can immediately meet the request. However, if the constraints, i.e. QoS levels, in a request cannot be met, it is necessary for the service provider and the client to reach an agreement, which can be achieved through negotiation, by altering the QoS parameters in the request, sending the request to the service provider, and waiting for a reply. If the request cannot be supported by the service provider, the client may send another request. This process can be repeated and this, in total, comprises the negotiation. This negotiation approach can however be extended to support a counter-offer, and, subsequently, if the resultant negotiation cannot provide the resource required, a suggestion can be made on when the resource would be available.

A QoS negotiation is essentially a match-making process, between a client's desire for a service with QoS constraints and a service provider's matching resource capability. For example, a client may request constant QoS levels during the lifetime of a service session, such as a data transfer service transferring a data set from point A to B at a rate of 100 Mbps. However, during the transfer session it is possible that the requested bandwidth cannot be sustained. In this case, the client may either request a decrease in the requested bandwidth while the transfer service is active or terminate the service. Alternatively the service provider must find additional capacity to sustain the QoS demand. A QoS re-negotiation requests the increase, or decrease, of QoS levels while the service session is active. If a 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 must run an admission control check, treating the request as a new QoS negotiation, subject to approval, or rejection.

The QoS negotiation process involves *service negotiation* and *QoS negotiation*. Decoupling service and QoS negotiations improves system availability and flexibility; 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. The QoS negotiation model proposed in this thesis requires a service negotiation phase, with an optional QoS negotiation phase, for negotiating resource characteristics and QoS levels. Two mechanisms are envisaged to obtain resource characteristics and service quality. Either the client application explicitly supplies resource characteristics and QoS levels required, or it relies on a service profile stored in the QoS registry, as discussed in Chapter 2.

In the latter case – using a service profile – the service profiles are either obtained from the service provider, based on feedback provided by clients, or generated using prediction models such as that in Jarvis *et al.* (2003). Quality levels within the service profile are dynamically updated and stored in the QoS registry. The service profile is for use by the QGS where a client specifically requests a service with its default QoS specifications, or does not have details on the resource configuration required to support the requested QoS level.

4.9 Quality-of-service Negotiation Protocol

The three participants involved in a negotiation protocol are the client, the QGS and the service provider. The QGS is the coordinator of the negotiation process between a client and a provider. The provider delegates the QGS to act on its behalf. There is no direct interaction between the client and the provider during negotiation.
The QGS supports a number of operations for use by a client, which include: ***Query***, ***Reserve***, ***Update*** and ***Cancel***, using an interaction based on an XML message exchange, with these operations explained in sections 4.9.1 to 4.9.4. The XML schemas for these operations are new and specifically designed for the G-QoSm architecture.

4.9.1 – Query

The QGS maintains, in a registry service, information about services and resources available to clients. The Query operation is used to interrogate the registry to find a service with particular QoS attributes. If a suitable service is found, the QGS reserves the resource(s) for a limited period (as 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.

```
<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:attribute name="computeImportance" 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:attribute name="networkImportance" type="xs:integer" use=" optional "/>
        </xs:complexType>
              </xs:element>
           </xs:sequence>
        </xs:complexType>
</xs:element>
```

Figure 4.8: XML Schema Definition for the Query Operation

Figure 4.8 is the XML schema definition for the Query operation with the required, and optional, elements as follows:

- **Service Name:** name of the requested service and its required element.
- **Service Type:** type of service, such as compute or network service, which is an optional element.
- **Temporal QoS:** concerned with the start and end time of the requested service, this is a required element associated with the two attributes: start time and end time.
- **Compute QoS:** describes the QoS attributes for the compute service, which are: 'capacity' (a required attribute), 'node count' (an optional attribute, as

the default is one compute node), and 'compute importance level' (an optional attribute), to specify the importance level as discussed in Chapter 3.

- **Network QoS:** describes the QoS attributes for the network service. These attributes are, 'source IP', 'destination IP' and 'bandwidth', which are all required, together with 'network importance level' (an optional attribute), to specify the importance level.

4.9.2 – Reserve

After a successful Query operation, and while resources are being held on a temporary basis, the Reserve operation is used to confirm the reservation. The QGS changes the status of temporarily-reserved resources to permanent, establishes a SLA and return an *agreement handle* to the client for use during service invocation. A schema for this Reserve is given in Figure 4.9 on the next page.

The reserve operation confirms a previously-made query for a service, with the reply including an agreement handle; a unique identifier for the requested service and its QoS information. The reserve schema has only one element:

- **Service Offer:** with only one attribute – query handler; a required attribute for confirming the reservation.

```
<?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 4.9: XML Schema Definition for the Reserve Operation

4.9.3 – Update

The update operation is used for re-negotiation in a situation where a client, during an active session, wishes to modify the constraints on particular QoS attributes. If the constraints are being relaxed, i.e. the QoS levels are reduced, then the operation is *guaranteed* to succeed. However, if additional resources are required then the request is treated as a new request, and the admission control procedure is applied, with the request either being approved or rejected. This is equivalent to a Query operation followed by a Reserve operation. Figure 4.10 shows an XML schema definition for the Update operation.

```
<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 4.10: XML Schema Definition for the Update Operation

The schema for the Update operation requires the following elements:

- **Agreement Handle:** an element returned from a previously-made reserved operation; used to reference the previously-made SLA; a required element.
- **New Temporal QoS:** for the re-negotiation, during the active session of the service. Needed to extend, or decrease, the service session period, with this element concerned with the new end time of the service. This element is only required if the end time of the service changes.
- **New Compute QoS:** where a compute QoS specification is re-negotiated, the capacity attribute is required. The number of nodes is optional as the default is one.
- **New Network QoS:** where the network QoS specification will be re-negotiated, the bandwidth attribute is the only one requiring updating, and is therefore required.

4.9.4 – Cancel

The Cancel operation, with schema given in Figure 4.11, cancels an agreement handle returned by a Reserve operation – i.e. it cancels a reservation. It may only be used before the service session starts. If the session has started, a different operation, not part of the negotiation process, may be used to release resources as part of the clearing phase of the QoS management function, as discussed in Chapter 2, namely the *service_completion* primitive part of the QGS API – with further details given in Chapter 5.

```
<?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 4.11: XML Schema Definition for the Cancel Operation

The cancel operation cancels a previously-made reservation, and therefore needs only one parameter:

- **Agreement:** contains one attribute, the agreement handler; a required attribute for cancelling the reservation.

Figure 4.12 is a sequence diagram for QoS negotiation protocol; it makes use of the four basic operations, namely; *Query, Reserve, Update* and *Cancel*, to implement the QoS negotiation, and re-negotiation, of a QoS session. The sequence diagram defines the general syntax of the protocol as follows:

1) The client/application sends a Query operation, i.e. initiates a negotiation request.
2) The QGS replies with a query handle, which is a reference for the query, only supplied if the query can be satisfied.
3) If the client/application accepts the offer, the client/application should use a Reserve operation, supplying the query handle to confirm the acceptance of the offer, and subsequently, the SLA is established.
4) The QGS replies with an agreement handle; a reference to the SLA.
5) Before the service, i.e. the QoS session has started, the client/application can use the Cancel operation to cancel the established SLA.
6) The QGS replies with the agreement status, i.e. whether or not the established SLA has been cancelled.
7) During the active phase, i.e. the QoS session, the client/application can use the Update operation to re-negotiate the established service agreement. For example, by requesting more resources, relaxing the resource specifications, or altering the end time of the service.
8) The QGS replies with a re-negotiation status, to indicate whether or not the re-negotiation has been successful.

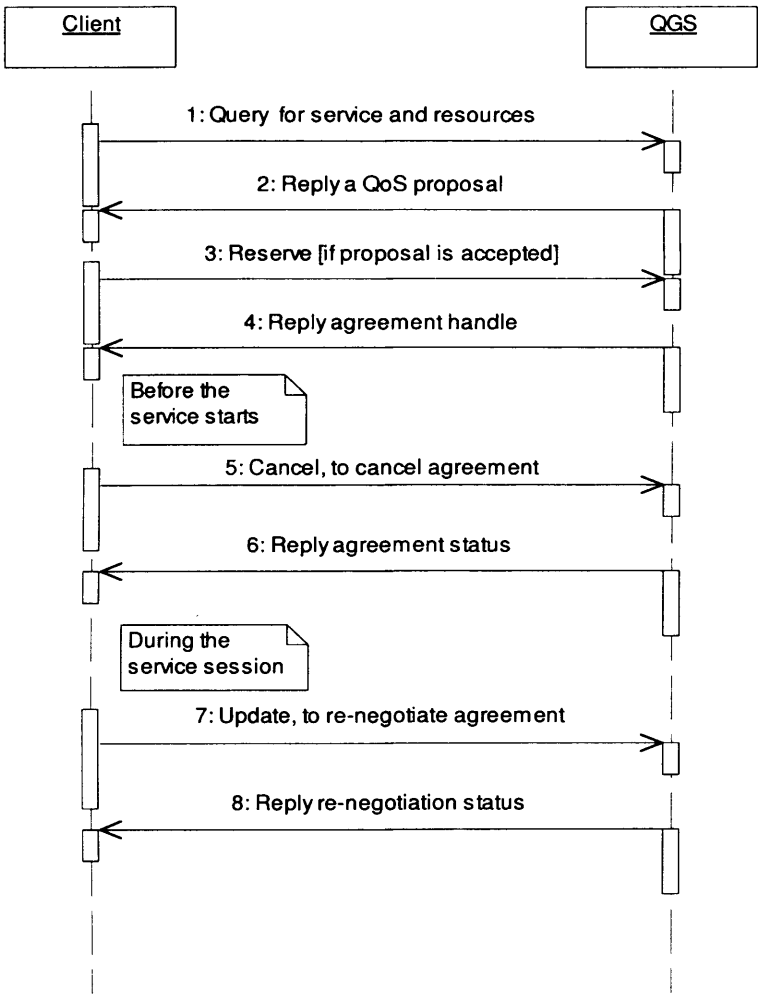

Figure 4.12: Sequence Diagram for QoS Negotiation Protocol

4.10 Summary

The G-QoSm architecture is presented in this Chapter. The basic building block of the architecture is the QGS; a QoS management system encapsulated into a grid service. This QGS manages physical grid resources to provide QoS functionality, such as resource reservation and allocation. The QGS can be published to a QoS discovery system, making it convenient for service discovery based on QoS properties. The QGS can further be used by a variety of QoS-brokering approaches, such as a hierarchical organisation of brokering agents.

G-QoSm is modular in design, giving flexibility for incorporating new resource managers as they become available. The architecture is a self-contained QoS management system and is built as a back-end service to the Java CoG Kit. This gives the G-QoSm further flexibility, as Java CoG is popular in the gird community, and many grid applications already use the Java CoG Kit to access grid back-end services, such as GT2 and GT3. Consequently, a CoG-based grid application has a natural transition into G-QoSm, and a new grid application can easily become QoS-aware via the API provided.

The process of QoS negotiation is presented, including a description of the protocol for message exchange between client and the QGS. The protocol is based on four message operations, *Query*, *Reserve*, *Update* and *Cancel*, which are conjectured to be suitable for QoS negotiation in a distributed system.

Chapter 5 ~ The Prototype discusses implementation aspects of the G-QoSm architecture, presenting a prototype and highlighting its key features, and demonstrates how a grid application can become QoS-aware, via the Java CoG API and the G-QoSm QoS Handler.

Chapter 5 ~ The Prototype

In this Chapter, implementation details of the G-QoSm prototype are presented, describing how the underlying resource managers are integrated into G-QoSm, and how a typical grid application uses the system.

5.1 Synopsis

A novel feature of the G-QoSm system is its implementation as a grid service within the GT3 toolkit. Being a grid service this allows G-QoSm to leverage services from Globus middleware such as security and the standard job submission mechanism, through GRAM, and other grid middleware services. The Java CoG kit (von Laszewski *et al.* 2001) client API library is extended to support access to the G-QoSm system, making use of services from the GT3 grid middleware. The prototype implementation of G-QoSm is an open-source implementation and can be downloaded and used.[1] The Java CoG Kit and Globus toolkit can also be downloaded from the Globus Alliance Web site (The Java CoG Kit Project, 2004; Argonne, 2004).

5.2 Implementation Overview

The implementation uses Java for most components, and C is used for creating a wrapper between QGS and the underlying resource managers, such as DSRT. Java allows for object-oriented design, modularity in system design, easy integration with other Java, C and C++ components and availability of APIs, to use the protocols for distributed computing and WS, such as SOAP and Web Services Description Language (WSDL).

Figure 5.1 presents an overview of the implementation architecture, showing how the QoS management component is implemented as QGS grid service. The QGS is deployed into the OGSI container within Globus GT3, with the entire GT3

[1] Appendix B gives the installation procedure for the QGS service and the computation resource manager.

middleware installed on a Linux-based machine. The grid node is identified as that machine which can offer its resources for use by G-QoSm clients. Clients may interact with QGS in two ways, either by directly using the API of QGS to negotiate a SLA request, or by using the API of the client library from the Java CoG kit, which provides most of the functions, such as negotiating the SLA. Using either way of interacting with QGS, the client must specify the allocation strategy for the chosen resources, whether time-domain or resource-domain, which are defined in Section 4.6. Once a client has negotiated a SLA request, the corresponding resource manager interface is configured accordingly, and the underlying resource manager is duly given the SLA parameters for actual resource allocation.

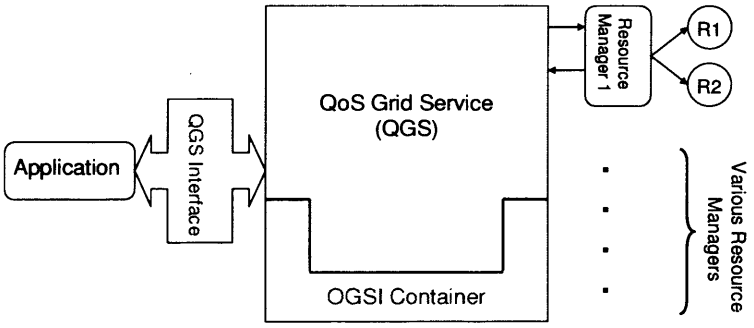

Figure 5.1: Prototype Implementation Architecture

5.2.1 – QGS Reservation Manager

The reservation component within QGS plays a major role in providing resource QoS provisions. Once a request is received from an application the functional requirements needed for the reservation are extracted from the request and formulated as resource specifications. These resource specifications are then submitted to the Reservation Manager with the request passing through a validation and admission control process; if the request is successful, a *reservation handle* is returned. This handle can later be used to claim or modify the reservation. In addition to implementing the admission control procedure, the validation function captures policy information necessary to validate the service request – for example, to discover any limitations on resource utilisation per service, or the class of service requested. The reservation manager, in general, performs admission control on reservation requests after a validation process has been undertaken by the policy

manager. Here the generic reservation component dynamically binds the reservation to a specific resource type, such as a network or a computing resource. In Appendix E, a Java class shows the reservation data structure and method used for resource reservation. Section E.1 shows a Java class for the reservation agent, which implements the reservation manager functionality. Section E.2 is a Java class for validating reservations requests.

5.2.2 – QGS API

Appendix A gives a WSDL specification of the QGS service interface provided by a set of APIs, including the specification of its operations. The term *application* is used to denote a client.

An application may interact with QGS in two ways:

- It can interact directly through the QGS API; this requires some extra handling by the application, such as using the API to negotiate SLAs as described in Figure 4.12, (i.e. using the negotiation protocol), or using the security infrastructure of the Globus API. This approach is ideal for building brokering services that use the QoS management entities.
- An application can interact with QGS via the Java CoG kit client library. Using this approach, the Java CoG kit is extended with a library for QoS handling. This extension provides: (i) compatibility with other services supported by Java CoG, such as a file transfer service and a job submission service, making it relatively easy to build a complex application; (ii) access to the Globus security infrastructure; (iii) the advantages of the built-in SLA negotiation component – the application submits the request and the Java CoG QoS handler is delegated, on behalf of the application, to undertake the negotiation phase; it returns a *null* response if it is not possible to establish a SLA and a SLA *identifier* (*SLA-ID*) response otherwise.

```
package org.globus.cog.qos.server.impl;
public interface Qos {
    public String service_request(String request);
    public String delete_request(String deleteRequest);
```

```
    public String sla_acceptance(String acceptanceRequest);

    public String sla_rejection(String rejectionRequest);

    public String service_execution(String executionRequest);

    public String service_extention(String extentionRequest);

    public String service_completion(String completionRequest);

    public String isResourceAvailable(String request);

    public String print_reservations();

    public String isJobCompleted(String id);

    public String setGramContact(String gramContact);

    public String deleteReservationEntries();
}
```

Figure 5.2: Main QoS Interface Class with Primitives for the QGS API

Figure 5.2 is a Java interface class that includes primitives for the QGS; the primitives, with a brief description, are:

- **public String service_request(String request):** sends a service request to QGS with the service name, allocation strategy, start and end times and service type, thus implementing the Query operation – Section 4.9.1. The request is encoded as XML attributes. A reply is returned, either with a service offer or with no offer. If a service offer is returned QGS has found suitable resources and temporarily reserved these. These resources await application approval so the temporary status can be changed to permanent, or until a pre-defined time elapses.
- **public String delete_request(String deleteRequest):** removes a reservation entry from the reservation table. After a **service_request** has been successfully completed, and a SLA has been established, the associated application has the chance to cancel the SLA. thus implementing the Cancel operation – Section 4.9.4. This feature is particularly useful when the application cannot use the promised resource due to some problem on the application side, for example, the application 'hangs' and cannot run at this time.
- **public String sla_acceptance(String acceptanceRequest):** accepts a SLA offer generated after a **service_request** had been successfully completed, and

the requested resources temporarily reserved. This changes the reservation status to permanent, thus implementing the Reserve operation – Section 4.9.2.

- **public String sla_rejection(String rejectionRequest):** rejects a SLA offer that was generated after a **service_request** had been successfully completed and the requested resources temporarily reserved. A SLA offer can be rejected for various reasons, such as the offer not matching the initial request or the application deciding to negotiate for more, or fewer, resources.
- **public String service_execution(String executionRequest):** activates a successfully negotiated SLA for a job submission with QoS properties. The executable files, data input/output files and the job submission mode – batch or interactive – are specified in the input parameter.
- **public String service_extention(String extentionRequest):** initiates a QoS re-negotiation during the active phase of service, i.e. the QoS session, thus implementing the Update operation – Section 4.9.3. A request to update an established SLA is passed to the reservation manger and, in particular, for the admission control procedure and validation function for the QoS levels to be increased. Such a request is automatically granted if the QoS level is to be reduced.
- **public String service_completion(String completionRequest):** releases resources when a service completes prematurely – i.e. before the SLA expires – thus starting the clearing phase as mentioned in Chapter 2. QoS management systems usually hold resources until SLA expiration, unless otherwise requested by the application.
- **public String isResourceAvailable(String request):** used by a brokering service to reserve multiple resources from more than one grid node. This allows an application to check whether resources are available without actually reserving them. This primitive is usually used before a **service_request** call.
- **public String print_reservations():** used by a brokering service, or system administrator, to query the reservation table and view all established SLAs and their corresponding reservation details.

- **public String isJobCompleted(String id):** used for notification purposes on a previously submitted job. The GRAM *gatekeeper* is contacted for the status of the submitted job: *running*, *suspended*, *completed* or *failed.*
- **public String setGramContact(String gramContact):** used when QGS is started, to supply the GRAM *gatekeeper* contact address, provided by Globus middleware for job submission management and control. All submitted jobs are processed by this specific GRAM *gatekeeper.*
- **public String deleteReservationEntries():** used by a brokering service, or a system administrator, to clear the reservation table. All reservation entries within the reservation table are removed. This is useful before shutting down QGS, or for testing purposes.

5.2.3 – Resource Manager Integration

The integration of a Resource Manager into G-QoSm requires the design, and implementation, of a software interface module specific to that Resource Manager. Such a software module, sometime called a *wrapper*, interacts with the Allocation Manager module in G-QoSm, and acts as a gateway to, and from, the Resource Manager. It translates requests from QGS into requests understood by the corresponding Resource Manager. Requests can include:

- return resource status and availability
- allocate resources
- de-allocate resources
- set resource allocation options and strategies

In Appendix C, the DSRT wrapper API is shown for a computational Resource Manager. Section C.1 shows a Java class for executing commands by the Resource Manager, such as the command to allocate resources. This modular design for G-QoSm, with a wrapper specific to the Resource Manager, allows flexibility in integrating new resources managers as they become available. To integrate a new Resource Manager, a corresponding wrapper implementation is necessary.

Most RMs provide some functions already provided by QGS, such as resource reservation; this duplication allows for flexibility. For example, suppose the network resource manager has two SLAs at the network level, denoted $SLA_{network1}$ and

$SLA_{network2}$ to distinguish them from SLAs for other resources. One can then define two resource capacities – i.e. pools of resources – one for *guaranteed* clients and one for *best effort* clients. One can map the G-QoSm view of the resource pool to the physical resources managed by a resource manager, such as a $SLA_{network}$ in this case. Such mapping allows a degree of flexibility and is consistent with the adaptation strategy of the G-QoSm model outlined in Chapter 3. This flexibility lies in the ability of QGS to manipulate the logical resource pool and conduct admission control checks whilst not actually committing physical resources until necessary. Figure 5.3 shows a model of resource manager integration in G-QoSm.

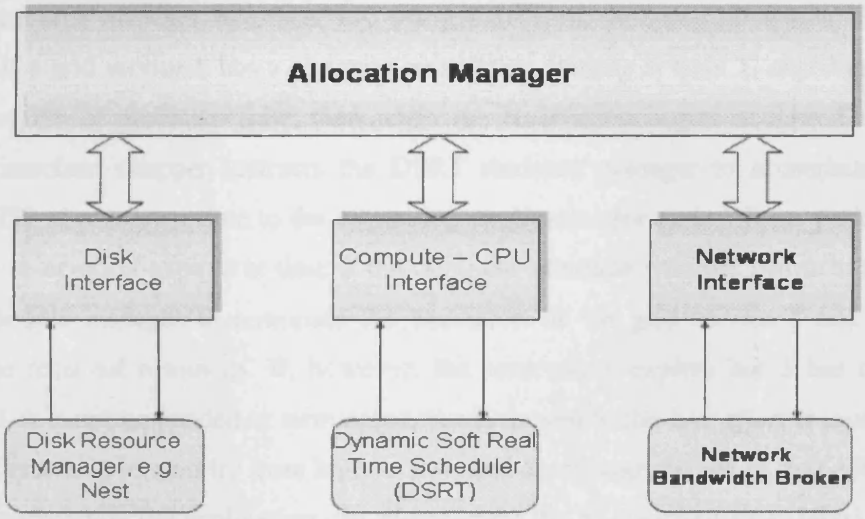

Figure 5.3: Integration of Resource Managers in G-QoSm

5.2.4 – Compute Resource Manager

The *compute resource manager* in G-QoSm is DSRT, a user-level soft-real-time scheduler, based on the changing priority mechanism supported by Unix and Linux (Chu and Nahrstedt, 1999). The highest fixed priority is reserved for DSRT itself, and a real-time process admitted by DSRT is run under its scheduling mechanism. The real-time process can thus be scheduled to utilise a specific processor time. DSRT has a flexible scheduling mechanism; for example, a real-time process can be scheduled to run for 100 ms at every 1000 ms interval. Consequently, the wrapper in G-QoSm, which interacts with DSRT, translates the application requests for processor time into a DSRT scheduling request. From an application point of view, the computing QoS supported by DSRT is specified in terms of a processor

percentage; for example, a real-time process requests 40% of processor time, which the wrapper translates to 400 ms of every 1000-ms interval.

The DSRT scheduler supports immediate reservations for an indefinite period. Although immediate reservation is a sound approach for reserving resources, immediate reservation for an indefinite period is not desirable, as outlined in Chapter 3, Section 3.3.1. Advance reservation, with a defined period, is more consistent with G-QoSm. To overcome this problem, the generic reservation module supported by G-QoSm manages advance reservation bookkeeping at the logical level, and the allocation manager implements resource allocation at resource manager level, via the specific resource manager interface, i.e. using DSRT for resource allocation. For example, if a grid service S has a compute reservation starting at time *X*, expiring at time *Y*, for *Z%* of processor time, then when the reservation begins at time *X* the compute interface wrapper instructs the DSRT resource manager to immediately schedule *Z%* of processor time to the requesting application for an indefinite period. When the reservation expires at time *Y* the compute interface wrapper instructs the DSRT resource manager to terminate the execution of the grid service *S* and to release the reserved resources. If, however, the reservation expires but *S* has not completed, it is not suspended or terminated, but is moved to the *best effort* resource pool, thus reducing its priority from high to low, and *S* continues to run in *best effort* mode. Alternatively, the application can re-negotiate the SLA before its expiration, or can negotiate a new SLA at expiry time.

When QGS receives a job submission request to be sent to the DSRT, the compute interface wrapper submits the request to the GRAM gatekeeper, which contacts the DSRT scheduler for actual job submission. Passing job submissions through GRAM utilises its services supported by Globus and its API supported by GRAM for job status monitoring.

5.2.5 – Network Resource Manager

The *network resource manager* (NRM) in G-QoSm, conceptually a DiffServ *bandwidth broker* (BB) (Teitelbaum *et al.* 1999), manages network QoS parameters within a given domain (generally defined to cover certain networks under the same administration), based on SLAs agreed at the network level between two domains, or

between a domain and a client. The NRM is responsible for managing inter-domain communication with NRMs in neighbouring domains to coordinate SLAs across domain boundaries. It may communicate with local monitoring tools to determine the state of the network and its current configuration. Figure 5.4 shows a BB-managed DiffServ domain.

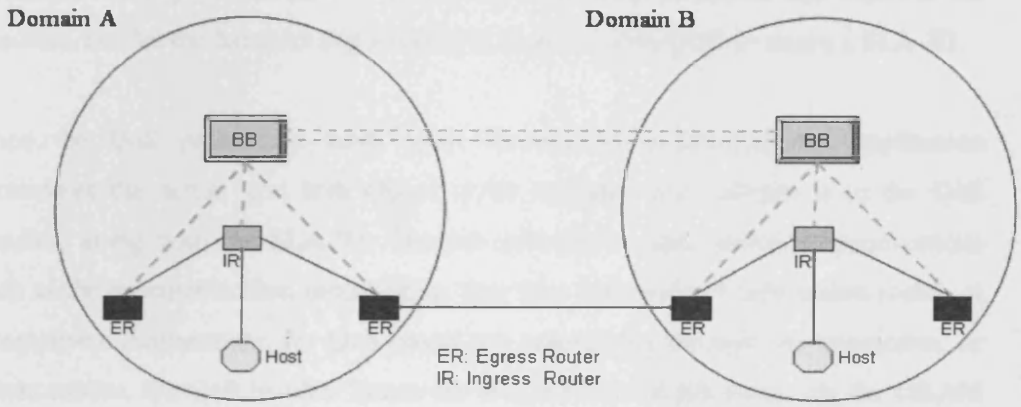

Figure 5.4: Role of Bandwidth Broker in DiffServ

The integration of the DiffServ BB into G-QoSm is similar to that of any other resource manager, as shown in Figure 5.3. A network resource manager interface is required to translate requests between the Allocation Manager of G-QoSm and the underlying network resource manager, the DiffServ BB. An application requesting network resources can use the same API provided by the QGS service; this API is consistent for the various resource managers integrated with G-QoSm.

An implementation of NRM called BB_{Basic}, from the University of New South Wales, (Sohail *et al.* 2003), is used in G-QoSm. BB_{Basic} supports most of the essential functions required to manage DiffServ domains. More details on the implementation and evaluation of BB_{Basic}, as integrated into G-QoSm, are presented in Chapter 6.

5.2.6 – Application Example using QGS

This section presents a scenario example of an application executing a QoS-enabled remote job submission to a grid node. The application developer must specify the QoS parameters for QoS negotiation. These parameters include start time, end time, resource type, and other QoS specifications such as allocation strategy, whether

resource, or time domain, and compute QoS requirements. Once the task object has been specified, the QoS Handler is delegated on behalf of the application to negotiate QoS requests; in this case, for compute resources. The QoS Handler is seen, from the QGS point of view, as a client. This is a useful approach particularly when the application requires more than one grid resource. All the application needs do 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 QGS to return a SLA-ID.

Once the QoS parameters have been successfully negotiated, the application formulates the actual grid task object to be executed and submits it to the QoS handler, along with the SLA-ID. The job submission task includes specifications such as the executable files, input/output data files and mode of submission (batch or interactive). Furthermore, for QoS-based job submission through the interactive, or batch, modes, the QoS handler listens for notifications of job status via the GRAM gatekeeper. This notification feature is important for some types of applications in keeping track of jobs which have completed.

The ease of use and benefits of using QoS properties can be demonstrated with an application. To enable other grid applications to use the QoS-enabled framework, a user needs to perform the following operations:

a) Create a task object, based on the Java CoG kit task object.
b) Depending on the type of required QoS function, set up the necessary objects for security, QoS functional specification and service access.
c) Instantiate a QoS Handler object.
d) Submit the QoS negotiation request task object to the QoS Handler.
e) Get a SLA-ID; for a successful submission.
f) Prepare the submission task along with the job specification, security context and service access.
g) Associate the created task with the QoS Handler object.
h) Submit the task object for execution.

Figures 5.5, 5.6 and 5.7 show Java code fragments demonstrating how an application can generate a QoS negotiation request, formulate a QoS-based job submission task and submit the formulated task object to the QoS handler. Appendix D shows a

complete working example with Java code for a QoS negotiation request and, in Appendix D.1 the Java code for submitting a QoS-based job.

```
/*** 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);

  // 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);
}
```

Figure 5.5: Formulating a QoS Negotiation Request Task

```
/*** 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);

  // 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);
}
```

Figure 5.6: Formulating a QoS-based Job Submission Task

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

Figure 5.7: Submitting a Previously Formulated Task Object to the QoS Handler

A *graphical user interface* (GUI) is included in the G-QoSm prototype to demonstrate the QoS functionality supported. The GUI proceeds through the steps outlined in the Java code fragments shown in Figures 5.5, 5.6 and 5.7.

Figures 5.8 and 5.9 illustrate how G-QoSm can be used to allocate processor resources with QoS specifications using a resource-domain allocation strategy. With this strategy, a certain capacity of the processor is reserved and the application submits jobs for execution within this reserved capacity. The process is implemented via the Java CoG kit API to create a task object, which is submitted to the QoS Handler for negotiation. If successful, a SLA-ID is returned for use in claiming a reserved resource.

A set of graphical components is included in the prototype to make access to QoS functions easier for non-technical users. Figure 5.8 shows a screen shot of the form used to specify the parameters of the QoS negotiation task to be submitted to the QoS Handler. Figure 5.9 shows a screen shot of the details of a QoS job submission object, specifying the executable application, called *mathAppl,* and a reserved processor time of 60%; mathAppl is a compute-intensive process and in this example is set to only use 60% of the total processor time. A simple feasibility study was conducted to evaluate the behaviour of the prototype system under heavy load, using compute-intensive processes that usually require full available processor time. Two compute-intensive competing processes were started before submitting the *guaranteed* mathAppl process.

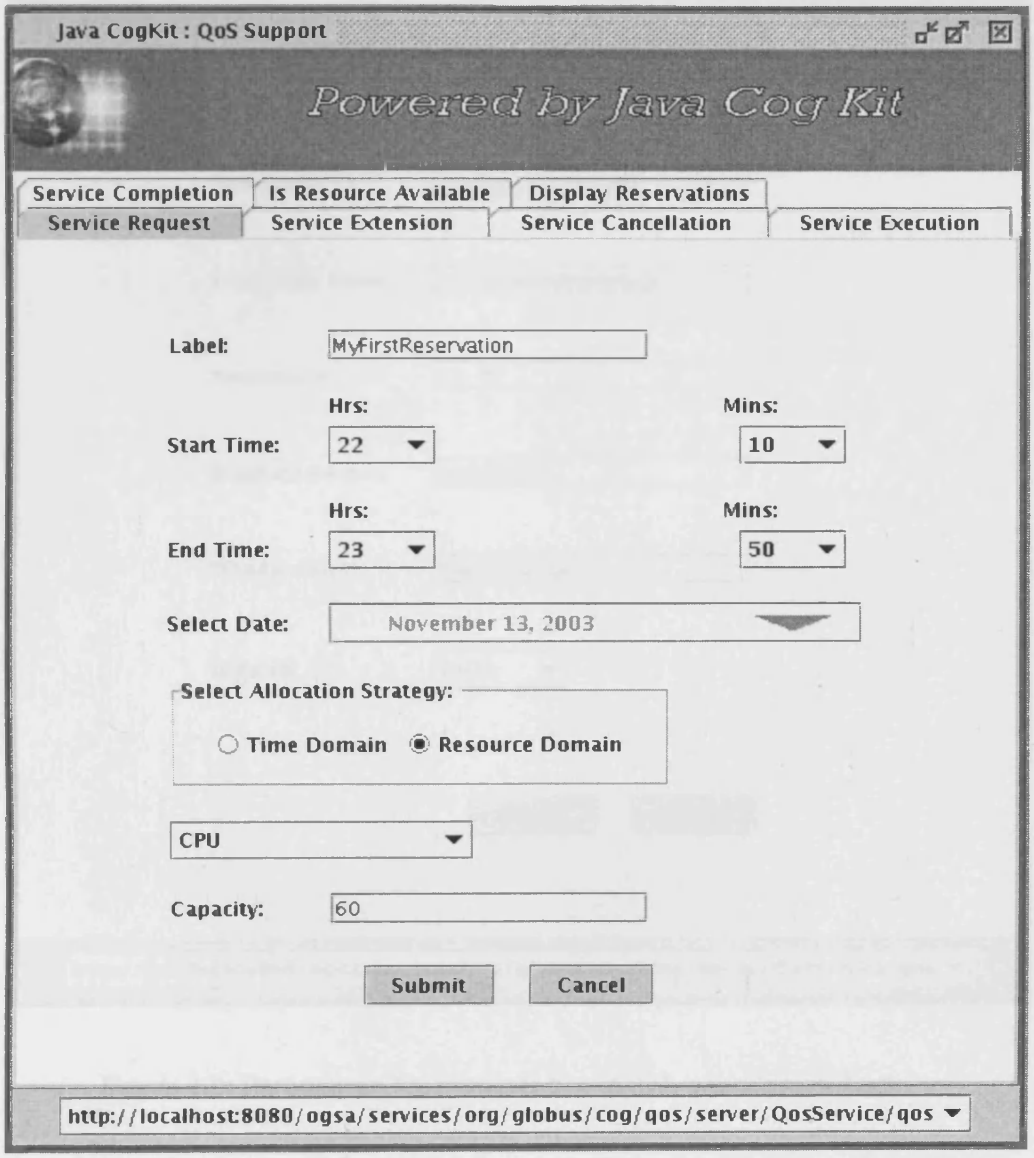

Figure 5.8: Parameters for the QoS Negotiation Task

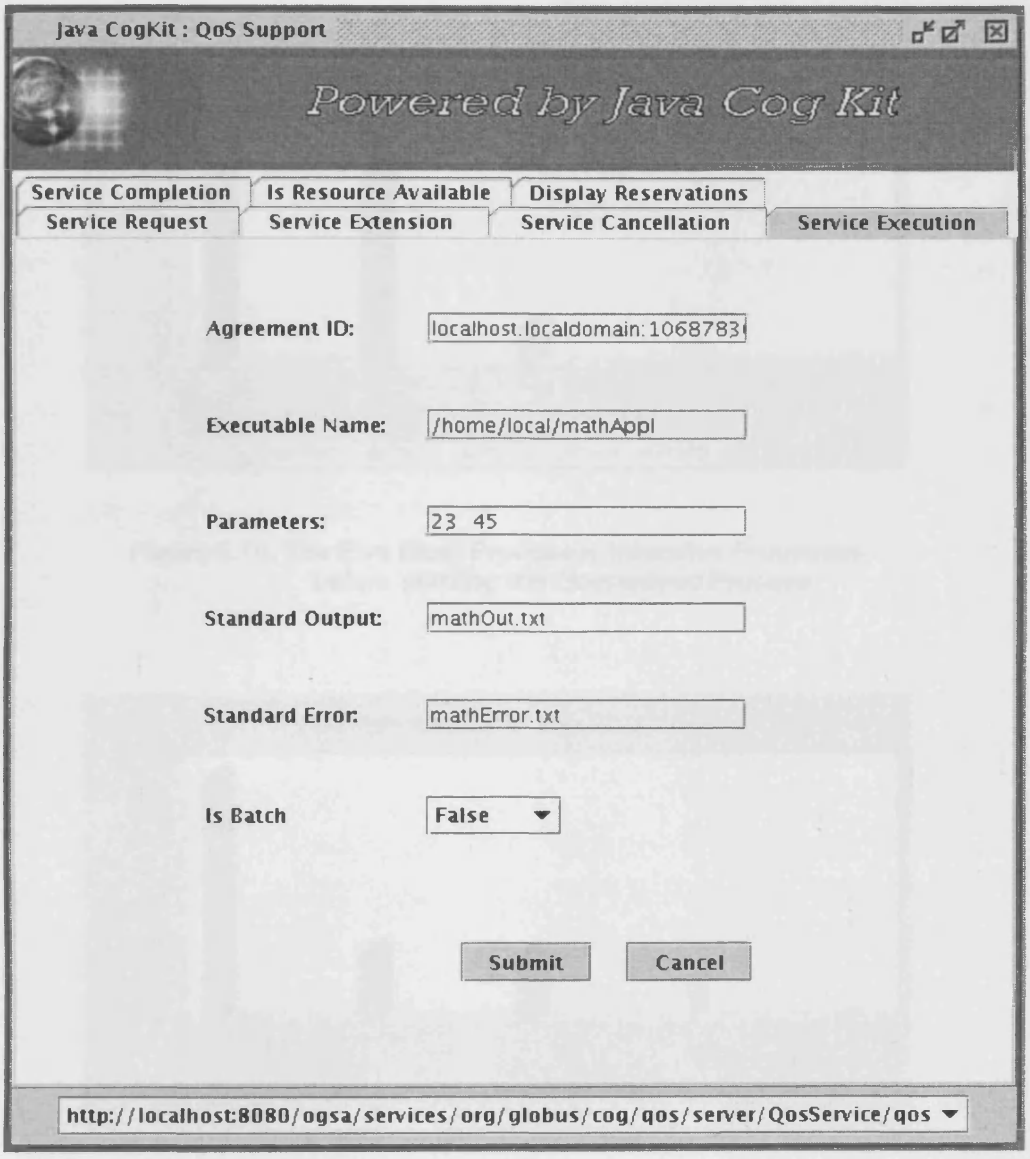

Figure 5.9: Parameters for the QoS-based Job-submission Task

A processor monitoring tool was developed to study the behaviour of processor utilisation during runtime. Examples of this monitor are given in Figures 5.10 and 5.11. In Figure 5.10, the five most processor-intensive processes are shown before mathAppl is submitted. Figure 5.11 shows the processor utilisation of the five most processor-intensive processes after mathAppl has been started; this Figure also shows mathAppl, as a *guaranteed* process, using 60% of the processor time of this grid node, with the competing processes using the remainder.

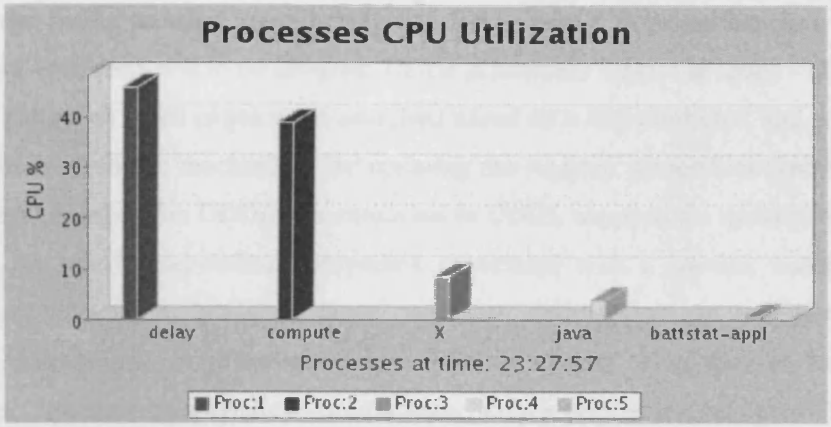

Figure 5.10: The Five Most Processor-intensive Processes before starting the *Guaranteed* Process

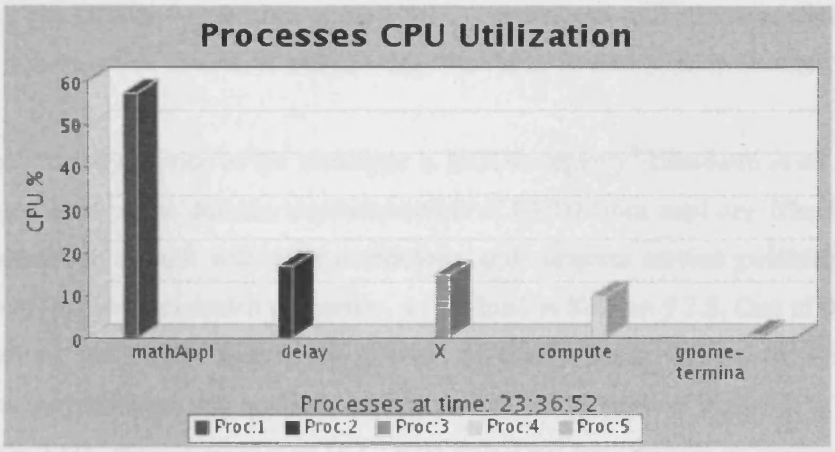

Figure 5.11: The Five Most Processor-intensive Processes after starting the *Guaranteed* Process

5.2.7 – QoS Registry Service

The QoS registry service is based on the Universal Description, Discovery and Integration (UDDI), which is a specification for distributed Web-based information registries for Web Services. UDDI allows HTTP-enabled business services to be published, and subsequently searched, based on their interfaces. UDDI consists of three components: 'white pages' to hold basic contact information and identifiers for

a company; 'yellow pages' to enable companies to be listed based on their industry categories (using standard taxonomies); and 'green pages' to record interface details of how a Web service is to be invoked. UDDI is however limited in scope – allowing white, yellow or green pages to be searched based on a few attributes, and does not provide an automatic mechanism for updating the registry, as services (and service providers) change. The UDDIe, an extension to UDDI, supports the concept of 'blue pages', to record user-defined properties associated with a service, enables the discovery of services based on these properties and support for *qualifier-based* search mechanisms as discussed below. UDDIe enables a registry to be more dynamic, by allowing services to hold a lease; a time period describing how long a service description should remain in the registry (ShaikAli *et al.* 2003).

The UDDI has four data types, for business and service information, which are XML-based data structures: *businessEntity*, *businessService*, *bindingTemplate* and *tModel*. The UDDIe – extension of the UDDI – makes use of the *businessEntity* and *businessService* data structures and provides the APIs, as described in Section 5.2.8.

The QoS registry service in the prototype is UDDIe registry[2] (ShaikAli *et al.* 2003), and based on a public domain implementation of UDDI from *uddi.org*. The UDDIe implementation is built with Java technology and supports service publishing and discovery, based on extended properties, as outlined in Section 5.2.8. One of the first applications for UDDIe was in the context of the G-QoSm framework, whereby services are published, and queried, dependent on QoS properties. Figure 5.12 shows a sample XML request submitted to UDDIe to search for services according to the specified QoS properties.

[2] The UDDIe registry service is available as open-source software from The Welsh e-Science Centre, Cardiff University. http://www.wesc.ac.uk/projects/uddie/uddie/download/

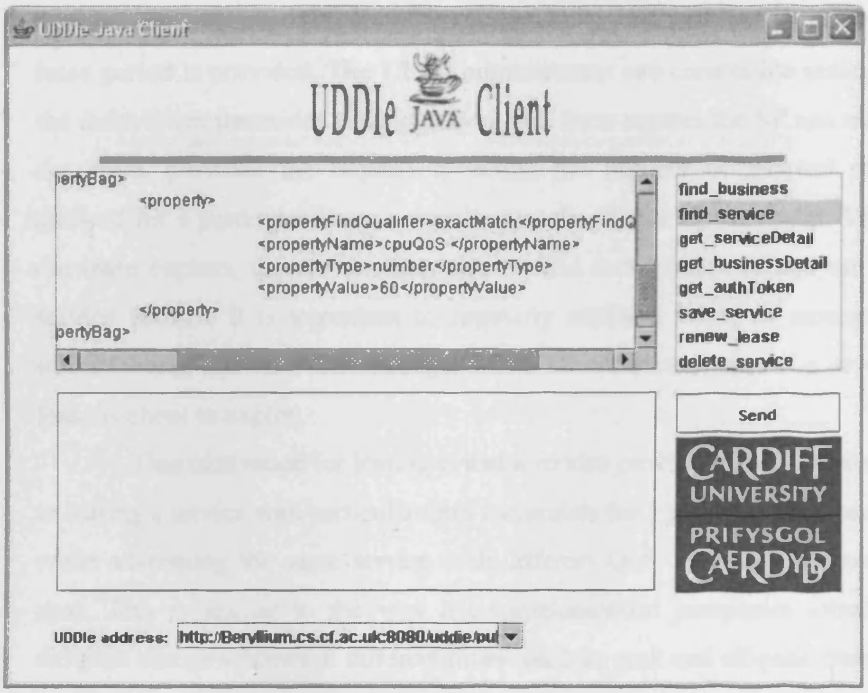

Figure 5.12: Sample XML Request Submitted to the UDDIe

5.2.8 – The UDDI Extension

Extensions in UDDIe comprise a set of *application programming interfaces* (APIs) for interacting with the registry system. These APIs are:

- **saveService:** used mainly for publishing service details. This API has been extended from the original UDDI system to introduce dynamic metadata for services. It is used to present QoS information, but can also be used to present various services' related information.
- **findService:** used mainly for inquiry purposes. In particular, this API includes queries based on information associated with services, such as service property and service leasing.
- **getServiceDetails:** used mainly for requesting more detailed information about services, such as *BusinessKey* and service information. This API includes service property information.
- **renewLease:** used by the UDDI administrator to control leasing information, and by the service provider (SP) to renew and set leasing information. Using the leasing concept, every service is associated with a lease, either for a limited, or an infinite, time period. The maximum number of infinite services is controlled by the operator; required to efficiently

maintain the registry. For a limited duration, a start and end date for the lease period is provided. The UDDI administrator can control the setting of the default, i.e. the initial leasing period. If a lease expires the SP can renew the lease, provided the request is within the number of renewal times allowed for a particular lease; controlled by the UDDI administrator. When the lease expires, the service becomes invalid and a client cannot use the service further. It is important to regularly renew a lease, or request an infinite lease, and an event manager alerts all connected users if a service lease is about to expire.

One motivation for leasing is that a service provider is often interested in leasing a service with particular QoS constraints for a particular time period, while advertising the same service with different QoS constraints at another time. This is similar to the way tele-communication companies introduce different charge schemes at different times, such as peak and off-peak charges. Another motivation is the introduction of grid service lifetime management in the OGSA specification, which specifies the validity of a service from creation to destruction.

- **startLeaseManager:** This set of APIs is used to monitor the lease constraints, by starting processes to monitor and delete expired leases from the registry. The UDDI administrator can control how often these processes are run.

In addition to these APIs, support for a *qualifier-based* search is included, to find services based on the value of a property specified by a qualifier expression, based on =, < or >. More complex expressions can be built using the logical operators **AND** and **OR**. These extensions to the UDDI registry and associated query mechanisms add search flexibility making UDDI useful for QoS-based systems.

Appendix F gives Java code for accessing UDDIe for services with QoS properties, and for selecting matching services based on the QoS property importance levels outlined in Chapter 3.

5.2.9 – Performance Experiments

An experiment was carried out to determine if the performance of the UDDIe registry is acceptable for applications requiring QoS provisions (Al-Ali *et al.* 2003d).

This experiment also aimed to find any bottlenecks in the query processing path. The experimental infrastructure includes the QoS manager, which processes clients' requests, the UDDIe registry and the database (used to store data related services, as well as service provider and user information), with the UDDIe and database on the same server. Queries were issued from another client workstation. The client workstation and server, located in the School of Computer Science at Cardiff University, were connected via a 100Mbps Ethernet network. The query *round trip time* (RTT) was measured as the time required for a query to be submitted from the client workstation, processed by the QoS manager and UDDIe, and the results returned to the client. Figure 5.13 shows a logical query path.

RTT can be computed as: $\text{QueryRTT} = \sum_{i=1}^{4} Ti$

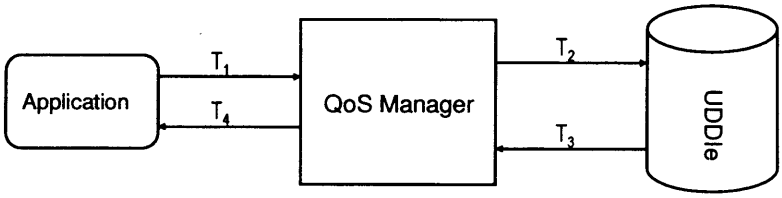

Figure 5.13: Logical Query Path

T_1: time taken to send the request from the application initiating the request to the QoS Manager including request processing at the QoS Manager,

T_2: time taken from the QoS Manager forwarding the request to the UDDIe, including the time taken at the UDDIe to process the request,

T_3: time taken from the UDDIe sending the reply to the QoS Manager and the QoS Manager regenerating the reply to the Application, and

T_4: time taken to send the reply from the QoS Manager to the Application.

The experiment comprised a mixture of queries for services with QoS properties:

- *Query1* requests services with QoS properties making use of the service property extension; the result is that no match has been found.
- *Query2* requests services with QoS properties along with service validity constraints, making use of the service leasing extension.

- *Query3* requests services with more complex QoS properties and leasing information. *Query3* introduces logical operations, making use of the range-based search mechanism.

The UDDIe registry was populated with a number of services and the three types of query were submitted. Each submission used a different service name and QoS attributes. Table 5.1 gives the average RTT for the queries submitted in each case.

Query RTT	*$Case_1$ RTT*	*$Case_2$ RTT*	*$Case_3$ RTT*
$Query_1$	2749	4421	5031
$Query_2$	9250	11469	13422
$Query_3$	9703	9407	10703

Table 5.1: Round Trip Time Responses
(in milliseconds)

The main purpose of the experiment is to show the performance obtained by integrating UDDIe into the G-QoSm framework. It was observed that the minimum time taken by the QoS manager and the UDDIe to process a request is about 5 seconds. If the list of services returned contains more than 30 services, and the QoS manager must choose between these based on the application's constraints then the response time is high. The maximum number of services returned was, therefore, restricted to five, which yields a better response time. The average response time for a successful request takes about 9 seconds; this response can still be improved by i) designing a more efficient algorithm to choose the best match, and ii) considering a hardware platform server with a higher specification than the experiment test-bed, and doing further experiments with, and without, the QoS Manager.

5.2.10 – Limitations

A limitation of the prototype is that one needs system administrator privileges to effectively configure QGS – i.e. root access on a Unix system. This is particularly true when configuring the underlying resource managers.

The prototype is Unix-based and was tested on Linux Red Hat version 9. Although the application can reside on any platform, QGS is restricted to a Unix system. Portability to other platforms is clearly desirable, but this restriction is because the

compute resource manager employed (DSRT), is a Unix-based implementation. The network resource manager integrated with the G-QoSm, namely BB_{Basic}, is a Java-based implementation which requires a Linux-based machine to be configured as a routing element for DiffServ support, with further detail on the BB_{Basic} integration given in Chapter 6.

5.3 Summary

The G-QoSm prototype – a QoS management service – is implemented as a grid service in the GT3 OGSI container. Such implementation enables QGS to make use of GT3 middleware services, for example, security and job submission through GRAM. The QGS provides an API, for client application and developers to interact with QGS, and uses grid resources with QoS provision. The communication protocol is based on the *de facto* Web Services protocol SOAP, and messages are encoded in XML. The core component within QGS is the reservation manager, which handles admission control, reservation validation and the generation of SLAs.

The reservation manager (the core component of QGS), manipulates logical entities that represent the actual physical resources. Such a manipulation is possible through the layered design of the resource manager integration architecture. The resource manager interface component is the actual entity that does the interaction between the allocation manager and the particular resource manager. The DSRT scheduler is used in the G-QoSm prototype as the compute resource manager. BB_{Basic} is used as the network resource manager, which supports DiffServ for networking QoS provisions.

This Chapter explains how a non-QoS-enabled application can be extended with QoS-based properties. To achieve this, the extended Java CoG API library and GT3 OGSI grid services container are used. Java code fragments demonstrate the use of the G-QoSm prototype.

The QoS registry service is based on the implementation of UDDIe, which has a number of extensions suitable for QoS-based discovery. The API of UDDIe is

outlined, and performance data is presented. Finally, some limitations of the current G-QoSm prototype are also presented.

Chapter 6 ~ Validation presents a verification of the compute QoS and network QoS support, and gives performance results for a grid application making job submissions, and undertaking data transfers with specific QoS requirements.

Chapter 6 ~ Validation

Certain classes of applications in grid computing, such as collaborative applications, must satisfy strict QoS constraints, as these application operate in collaborative mode and data must therefore be stored, processed and delivered over a limited time span – for example, tele-immersion, visualization and computational simulation. QoS management is required to plan and guarantee the timely interaction among components of such applications. To validate G-QoSm two example applications were chosen for performance analysis, one computation-intensive and the other communication-intensive. The first is an image processing task derived from a nanoscale structure application (Al-Ali *et al.* 2004b). The second involves the use of the DiffServ architecture with a Bandwidth Broker (BB) component (Al-Ali *et al.* 2004d).

6.1 Computation-Intensive Example

The G-QoSm prototype was used to manage a nanoscale structure application, being developed as part of Argonne National Laboratory's advanced analytical electron microscopy program (Zaluzec, 2004). 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 a researcher to study subtle changes resulting from micro-structural differences, such as electro-magnetic domain formation. The analysis of this data requires a resource-rich grid infrastructure satisfying real-time constraints. During an experiment, results need to be archived, remote computing resources need to be reserved, and the data must be moved to the computing resources for analysis. Moreover, results need to be gathered and presented in a meaningful, human-readable form.

The need for a reliable computing infrastructure is demonstrated by the simplified flow diagram in Figure 6.1. The elementary logic of the instrument control can be

expressed as a sequence of interacting processes: *Data Acquisition* gathers time-delayed images from the electron microscope; *Backup* stores incoming data; *Data Analysis* analyses the time-delayed images; and *Result Display* gathers the results from the data analysis, in a form suitable for interpretation and continuance of the experiment.

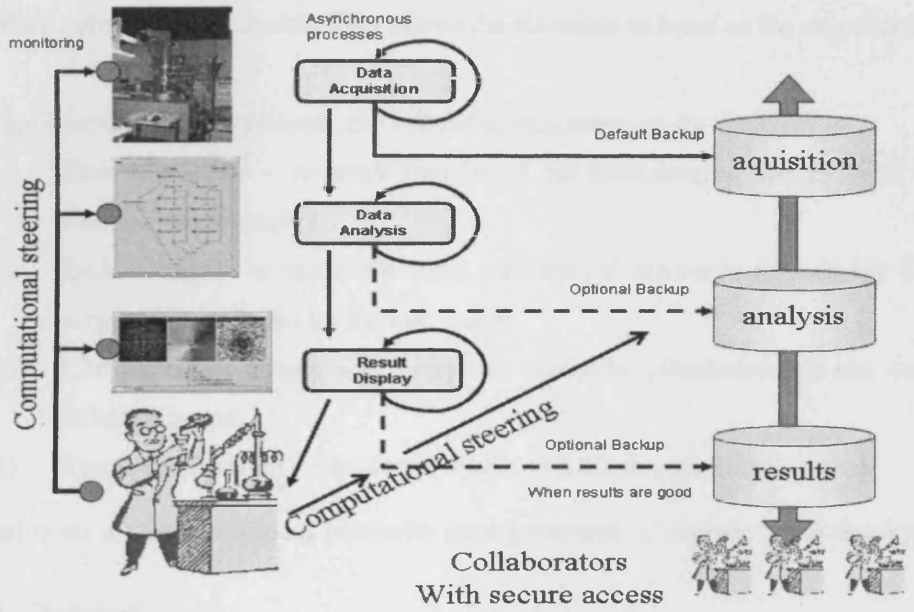

Figure 6.1: Asynchronous Processes in Nanoscale Structure Application

This nanoscale structure example exhibits a pattern typical of many scientific applications in high-end instrument scenarios. The pattern includes a high volume of interaction during an experiment which must be dealt with in an adaptive and flexible way. The instrument operator's interface with the grid must be as simple as possible, while at the same time providing flexibility to interactively modify the experiment.

The Java CoG kit provides a convenient abstraction for formulating tasks, such as file transfer, job execution and job management. At the same time, it hides much of the complexity from a grid application developer. The Grid Application Toolkit (GAT) interface, developed in the European GridLab project, also provides a generalised collection of calls to shield grid applications from implementation detail of the underlying grid middleware. GAT uses adaptors that facilitate the application

choosing a specific binding (from the GAT interface to the underlying technology) which implements a specific functionality (Taylor *et al.* 2003). Using a suitable interface, a scientist will be able to interact with the experiment resources and decide when, what, and where data gathered during the course of the experiment is backed up. Because of the focus on the experiment itself, the use of the grid should, as far as possible, be via abstractions, i.e. details of the grid should be hidden from the scientists doing the experiments. This allows the scientists to focus on the experiment.

The application example presents the following requirements for QoS:

a) Data acquisition – network transfer of the time-delayed images from the electron microscope;
b) Disk storage – to cache the large amounts of incoming data during data acquisition, and also for backup usage;
c) Computational power – to carry out scientific calculations on the time-delayed images;
d) Result presentation – transfer of results to a display for interpretation.

Experiments in this thesis focus primarily on requirement 'c' and the result display.

6.1.1 – Test-bed

The test-bed for the experiment included two Linux-based computers: one with a 1.8 GHz Pentium processor and 256 MB of memory, acting as the service consumer; the other, a 1.2 GHz Pentium processor and 512 MB of memory, acting as the service provider. All machines were connected through an Ethernet local-area-network. This experiment was not carried out on a wide area network, as one needs a super-user access privilege to install and configure the G-QoSm prototype. Deployed on these machines were GT3 OGSI service container, GT2, and the Java CoG kit. Experiments were carried out using two different approaches: one with a QoS handler through the Java CoG kit and the second with a GT2 handler through the Java CoG kit.

6.1.2 – Time-domain Allocation

The nanostructures image analysis task, based on a sample electron diffraction using up to 900 input images, was executed on the test-bed using a time-domain strategy

for resource allocation, as outlined in Section 4.6. With the entire compute node reserved for the application, multiple jobs were submitted to the reserved node but only one was executed, the job that had previously made a reservation.

Two sets of runs were conducted, one with job submission based on QoS and one with standard job submission based on GT2. In the job submission based on QoS the submission is done through the QoS Handler in the Java CoG Kit, and involves QoS management, such as resource reservation and SLA establishment. Each set consisted of two groups of four runs each for observation and analysis purposes. In the first group, four collections of images were processed in parallel, submitting the entire collection to the grid node for processing at the same time. In the second group, the same four collections of images were processed sequentially, submitting one image at a time to the grid node for processing. The four collections contained 25, 50, 75 and 90 images respectively. Figures 6.2, 6.3, 6.4 and 6.5 show the performance results relating the number of images and the time taken for each run.

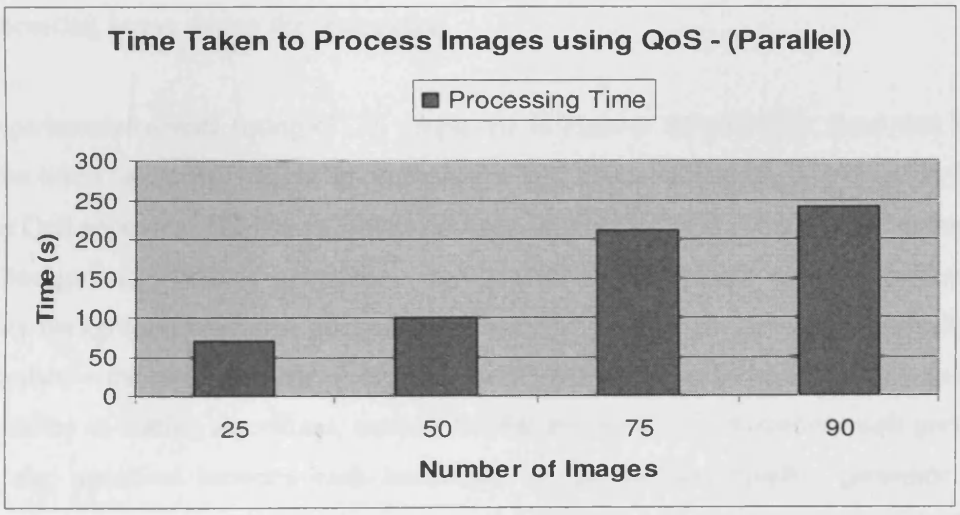

Figure 6.2: QoS-based Execution – Parallel

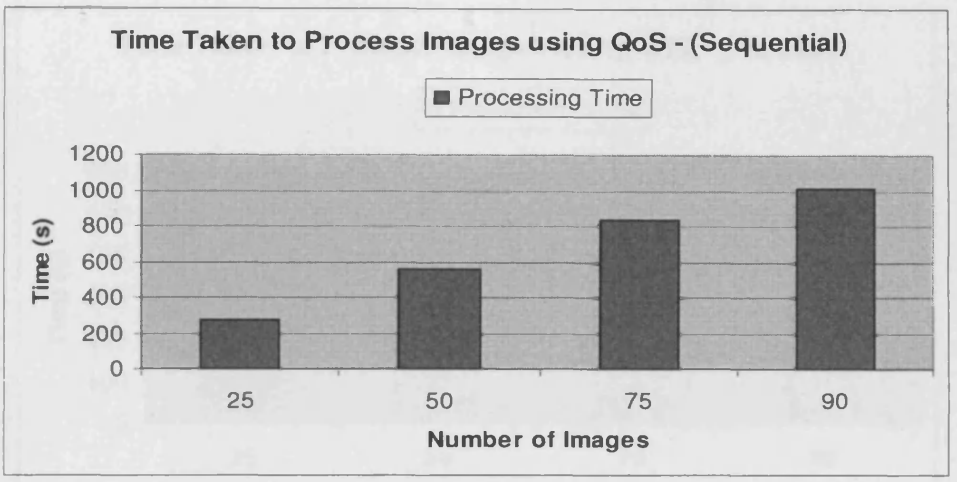

Figure 6.3: QoS-based Execution – Sequential

The results displayed in Figures 6.2 and 6.3, obtained from the QoS approach, show that the time taken to process the images, in both parallel and sequential mode, is less than for the GT2 approach. This is expected, since the reservation mechanism employed in this time-domain strategy reserves the entire processing power of the grid node for the QoS-based application, which prevents other processes from using processing power during the reservation.

Experimental results (using GT2), displayed in Figures 6.4 and 6.5, show that the time taken to process images in both parallel and sequential mode, is more than for the QoS approach. The reason is that multiple processing loads were applied through a background workload generator – to simulate a shared multi-user environment. This background workload generator is used to sort a list of up to 10,000 random numbers – the actual number of elements in the array is also picked randomly – using a variety of sorting algorithms, such as bubble and heap sort. A random wait period is also specified 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 managed by a processor. Because the GT2 technology does not employ a reservation mechanism, other processes can use processing power while the job submitted is being processed.

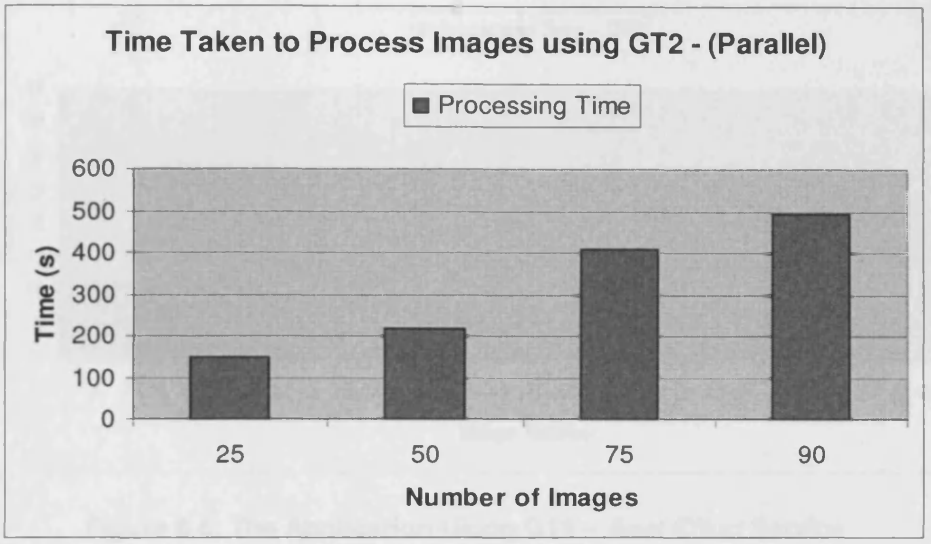

Figure 6.4: *Best Effort* Execution using GT2 – Parallel

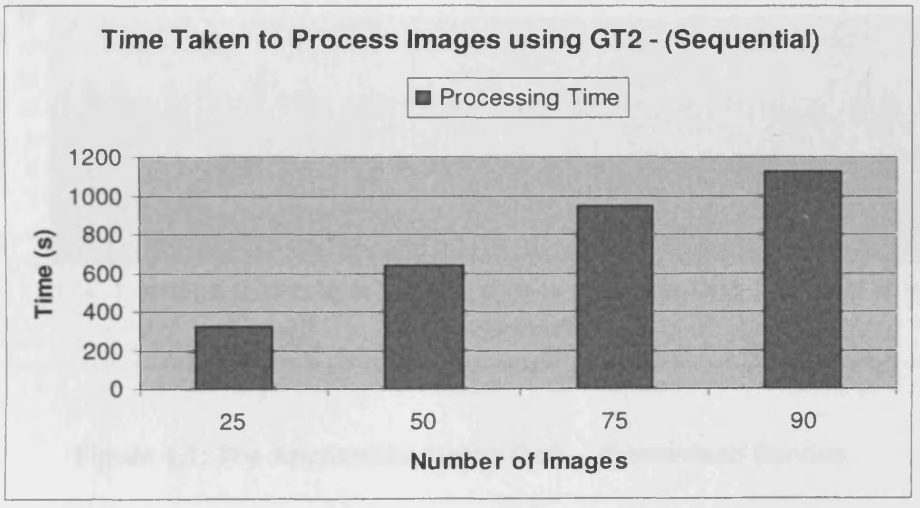

Figure 6.5: *Best Effort* Execution Using GT2 – Sequential

Figures 6.6 and 6.7 show results for the nanostructure application in GT2 and QoS, for, respectively, *best effort* service and QoS *guaranteed* service. Figure 6.6 indicates that processing time per image generally takes from 10 to 30 seconds. This 20 second variation in the image processing time is quite significant, compared to the variation from the QoS approach shown in Figure 6.7 and discussed in the next paragraph. The time variation from the *best effort* approach, in Figure 6.6 makes the processing pattern inconsistent.

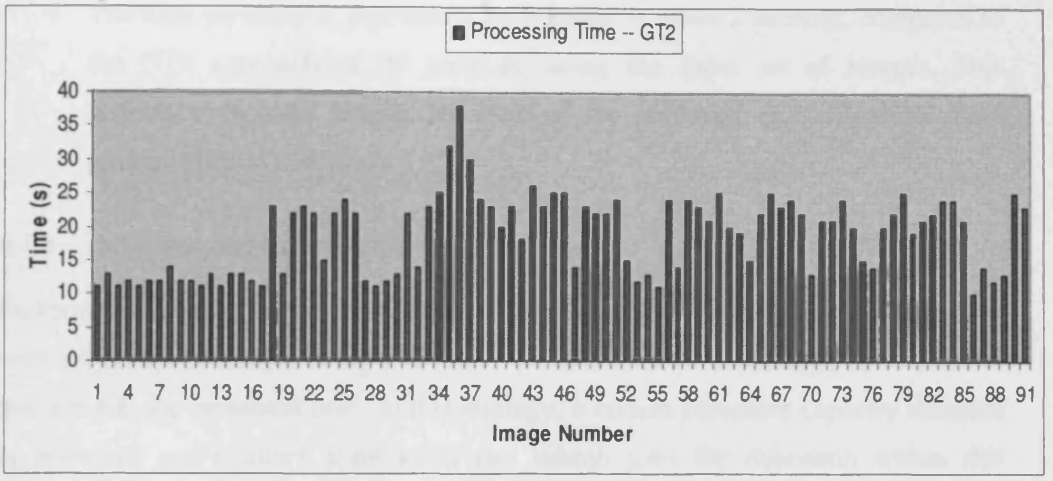

Figure 6.6: The Application Using GT2 – *Best Effort* Service

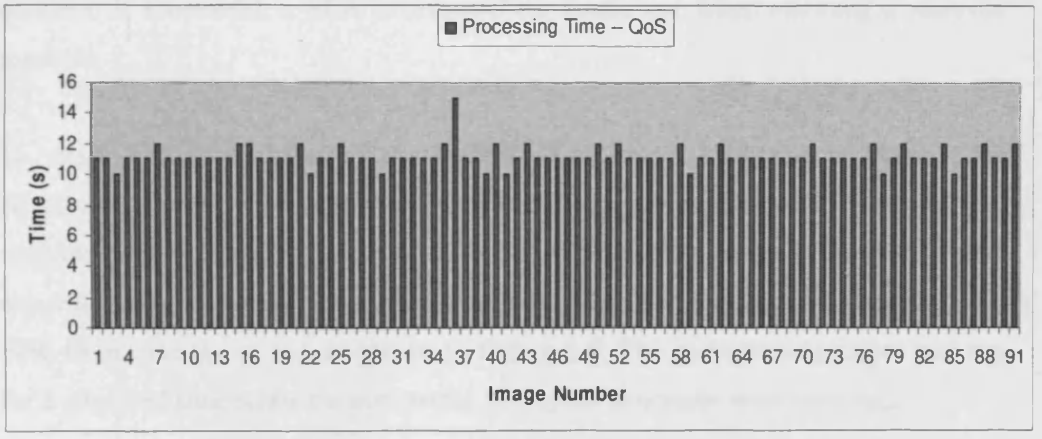

Figure 6.7: The Application Using QoS – *Guaranteed* Service

Figure 6.7, using the QoS *guaranteed* approach, shows an execution time per image ranging from 10 to 12 seconds, except for image number 36 which took approximately 15 seconds. The same image is shown to take approximately 37 seconds in Figure 6.6, based on the GT2 *best effort* mechanism, which indicates that image 36 has greater processing requirements than the other images. The variation in image processing time using QoS constraints is quite small, which makes the processing pattern reliably consistent. From the above results, one can observe that application processing using the proposed QoS approach provides the following advantages:

- The processing of the images gives 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, making the proposed QoS approach more predictable and consistent.

6.1.3 – Resource-domain Allocation

Performance results, using the G-QoSm framework to allocate processor resources with a QoS specification, using a resource-domain allocation strategy, as outlined in Section 4.6, are presented here. In this strategy, a certain processor capacity resource is reserved, and a client application can submit jobs for execution within this reserved capacity. The process is implemented using the Java CoG kit to create a task object which is submitted to the QoS Handler to negotiate the required resources or services. If successful, a SLA is returned for future use when claiming a reserved resource.

To evaluate the behaviour of the system under heavy load, and to observe the effectiveness of job submission with QoS constraints, two experiments were run, one with two processes run in *best effort* mode – i.e. without a processor reservation; and one with one process run in *guaranteed* mode – i.e. with a processor reservation of 60% from time (t_{25} to t_{65}), as shown in Figure 6.8. The *guaranteed* process was run for a specified time while the competing *best effort* processes were running.

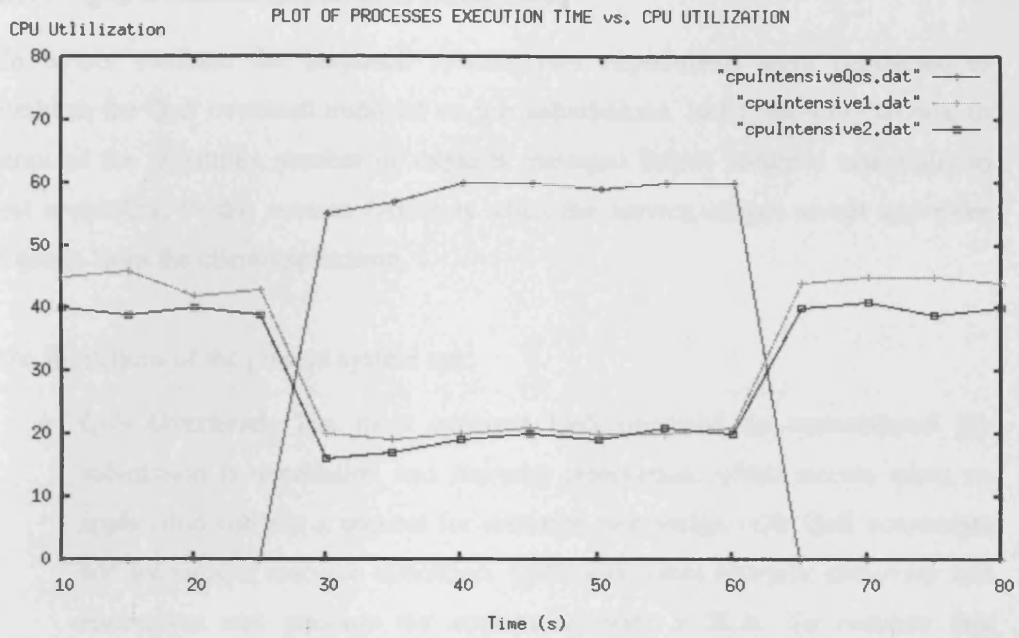

Figure 6.8: Execution of *Guaranteed* and Competing Processes

To further study system behaviour, and to observe the execution pattern of the *guaranteed* process, performance data was observed shortly before the *guaranteed* process started, then periodically every 5 seconds, until shortly after completion. Figure 6.8 plots the execution pattern.

- From t_{10} to t_{25}, two computation-intensive processes competed for 100% use of the processor.
- At t_{25}, the *guaranteed* process, with a *guaranteed* processor usage of 60% started, and lasted until t_{65} (based on processor reservation).
- From t_{65}, the two computation-intensive processes again competed for 100% use of the processor.

During the active session of the *guaranteed* process, the *guaranteed* processor usage of 60% was maintained, with the remainder of the processor shared between the other processes. At t_{65}, when the *guaranteed* process completed, the two computation-intensive processes started to compete for 100% usage.

6.1.4 – QoS Overhead and System Limitations

To further evaluate the proposed system, two experiments were conducted to establish the QoS overhead imposed on job submissions, and system limitations, in terms of the maximum number of requests managed before failure – essentially to test scalability. In this context failure is when the service cannot accept any more requests from the client/application.

The limitations of the present system are:

- **QoS Overhead:** The most apparent QoS overhead on conventional job submission is negotiation and resource reservation, which occurs when an application submits a request for resource reservation with QoS constraints and subsequent resource allocation. QGS undertakes resource discovery and reservation and presents the application with a SLA. To measure this overhead, an application generating (at various times) about 1,000 requests for QGS was monitored. The interval recorded was from the time the application initiated the QoS request until the request was acknowledged by QGS. The time taken to acknowledge QoS requests ranged from a best case of 50 ms to a worst case of 200 ms. The acknowledgement time depends on how busy QGS is and on the network connecting the client/application and QGS – in a wide area network infrastructure this acknowledgement time might differ due to the network factor. 50 to 200 ms is not significant compared to the time normally reserved for a QoS session, in the order of minutes or even hours; and this overhead is negligible.
- **System Limitation:** A test was conducted to determine scalability in terms of the QoS request load. A large number of requests, at different times, were issued by the client/application over the network. It was observed that QGS cannot accept more requests after approximately 3,600 requests in approximately 6 minutes, after which denial of service occurs, due to a hardware limitation on the experimental test-bed. The reason for this limitation was found to be the prototype system's reservation table, which contains information about reservations, agreements and SLAs. This table, maintained in primary memory, was found to be almost full when denial of service occurred. The denial of service could also have arisen from the

process table becoming fully utilised, as more requests were forwarded to the server. To overcome this constraint, it is planned to store the reservation table in a disk file rather than in the main memory, or store the reservation table in a database, such as Oracle or MySQL, for more efficient data retrieval.

6.2 Communication-intensive Example

This Section examines G-QoSm's network QoS support and provides experimental results (Al-Ali, *et al.* 2004d). The network QoS support is provided via the DiffServ architecture and relies on a BB component. Performance results, using a BB, along with other elements, in the G-QoSm framework are presented. BB_{Basic} implementation is integrated with the G-QoSm framework to provide network QoS. BB_{Basic} is University of New South Wales implementation of a BB (Sohail *et al.* 2003).

6.2.1 – BB_{Basic} Implementation

The BB_{Basic} is based on the concept of BB – background information is provided in Appendix G. The BB_{Basic} implementation provides the features of the BB architecture as outlined in Appendix G. It is implemented in Java and follows a client-server model. BB_{Basic} can interact with Linux-based routers, unlike the systems reviewed in Section 2.7, as the routing element, whereas Linux routers need to have DiffServ support enabled, which is built into the Linux kernel from version 2.4 onwards. Java handles remote client-server functionality through TCP sockets. A BB_{Basic} can handle multiple connections from the routers and clients simultaneously. The implementation provides a query facility, about resources and SLAs, for users and network administrators who can request details. Implementation details for BB_{Basic} are available in Pham and Nguyen (2003). Some relevant implementation details of BB_{Basic} are explained in the following Sections.

6.2.1.1 Inter-domain

The inter-domain protocol embedded in BB_{Basic} is designed on the specifications of *simple inter-domain BB signalling* (SIBBS) protocol, denoted here as $SIBBS_{Basic}$. The specification of SIBBS (QBone, 2002) does not explicitly state the mechanism that a BB uses to gather information about neighbouring BBs. Nor does it give detail about their administrative domains. $SIBBS_{Basic}$ collects this information from its

database, which contains a comprehensive network map, enabling BB_{Basic} to identify the neighbour which should be contacted to complete an application's *resource allocation request* (RAR). Whenever the resources requested include those from other domains, BB_{Basic} gathers information from neighbouring BBs and contacts them via $SIBBS_{Basic}$. A neighbouring BB checks its resources, and if the request is accepted, propagates it to the next BB in the direction of flow. The process continues until the request reaches the BB with the destination host in its domain, and replies are sent back in the reverse manner. After sending the *resource allocation answer* (RAA), in the case of request acceptance, BB_{Basic} configures its edge routers via the intra-domain protocol to allocate network resources for the accepted flow.

6.2.1.2 Intra-domain

Common Open Policy Service for Provisioning (COPS-PR) (Halim and Darmadi, 2000), the intra-domain communication protocol used in BB_{Basic}, is an independent implementation linked to BB_{Basic}. The COPS-PR and BB_{Basic} combination was tested on Linux routers, with results (Halim and Darmadi, 2000; Pham and Nguyen, 2003) indicating that BB_{Basic} effectively manages network resources by reconfiguring the relevant routers with COPS-PR when required. BB_{Basic} functions as a *policy decision point* (PDP) that connects to its own domain routers, at a *policy enforcement point* (PEP), to configure these according to a pre-defined domain policy. Whenever BB_{Basic} accepts a request, related core and edge routers (if required) are contacted via COPS-PR. A core router needs reconfiguration when it is a *first-hop* router for the flow; with reconfiguration required for marking and shaping the flow's packets. Marking of packets is required to classify the packet, and shaping is required to keep the flow below agreed limits. The edge router is contacted by the BB when the destination or source of the requested flow is in a different DiffServ domain, to enable the edge router to filter, shape, schedule, or mark the packets according to the SLA.

6.2.1.3 Database

A MySQL database is used to store information related to a BB. The information is divided into three parts: user, BB and network. The user part consists of an application's SLA, password and resource request information. The BB part contains relevant information about peer BBs, and the SLAs with these BBs. The network part contains information on the network, such as network domains and network addresses,

essential to determine the routers needing reconfiguration when a BB accepts a request. Network information is also necessary to find the neighbouring BBs to contact for resources acquired from multiple domains.

6.2.1.4 User/Application

BB_{Basic} has multiple interfaces for application access; these interfaces allow an application to choose the most suitable mechanism for interaction with BB_{Basic}. Distinct interfaces are provided, for example, a Java API and a Web-based client for administrators and users. Detailed description of these interfaces and information about their use is available in Pham and Nguyen (2003).

6.2.1.5 BB_{Basic} Integration

The BB_{Basic} integration into G-QoSm enables support for managing network resources. To integrate a new resource manager, it is necessary to specifically design an interface, as shown in Figure 5.3.

6.2.1.6 Network Interface

The network interface does the translation of requests between the QGS and BB_{Basic}. The QGS may include four types of request:

- **Querying Resources:** Resource querying can be classified into querying a $SLA_{network}$ for information related to a specific $SLA_{network}$ ($SLA_{network}$ relates to SLAs between DiffServ domains), and querying the status of an RAR within a particular $SLA_{network}$., with a RAR corresponding to a G-QoSm SLA (i.e. application/user SLA). Querying a $SLA_{network}$ allows the QGS to enquire about the capacity of a specific network element currently being used, and the remaining capacity available for use. The second type of query allows the QGS to enquire about the status of a particular established RAR, and to view associated information such as start and end times, network bandwidth granted, type of network service, such as *expedited forwarding* (EF), and source and destination IP addresses. EF is a mechanism used to build assured bandwidth in DiffServ domains, based on low delay, jitter and packet-loss rate (Jacobson *et al.* 1999).

- **Allocating Resources:** Resource allocation involves issuing a RAR associated with a pre-defined $SLA_{network}$. Parameters required include the amount of network bandwidth required, the type of network service, the associated $SLA_{network}$, start and end times, and source and destination IP addresses.
- **Releasing Resources:** The release or de-allocation of resources only works for pre-established RARs. Here, the RAR can be deleted – i.e. the removal of network QoS privileges – with the parameter, required for this request, the RAR *identification number*. This operation changes the network traffic service type from *guaranteed* service to *best effort* service, if the network resources are still needed. This is consistent with the G-QoSm concept that network flow will not be terminated, or suspended, but will rather be reduced to a low priority type service.
- **Modifying Requests:** Request modification affects a $SLA_{network}$ or a RAR. For example, a $SLA_{network}$ can be modified by changing its bandwidth capacity or the type of network service being provisioned. Similarly, a RAR can be modified to change its bandwidth capacity, or start and end times, i.e. implementation of re-negotiation requests, which is consistent with the G-QoSm concept.

6.2.1.7 Requesting Network Resources

With the integration of BB_{Basic} into G-QoSm, grid applications can request network resources with QoS constraints. The protocol is similar to that for computational resources, outlined in Chapter 5 and in Al-Ali *et al.* (2004a). G-QoSm extends the Java CoG kit architecture and makes use of its API, as discussed in Chapter 5. Figure 6.9 shows Java code for initiating a request for network resources; in particular, it shows a negotiation task for network resources. The API used is similar to that in Chapter 5, the only difference being the task attributes which should be specific to the type of resource under consideration, in this case the network; and attributes like network bandwidth, source IP and destination IP are expected.

```
/*** QoS: Prepare Negotiation Task ***/
private void prepareQosNegotiationTask() {
 // create a QoS service and setup QoS attributes for network  resource
 Task task = new QosTaskImpl(``myTask", QoS.NEGOTIATION);
 this.task.setAttribute("startTime", startTime);
 this.task.setAttribute("endTime", endTime);
 this.task.setAttribute("networkBandwidth",networkBandwidth);
 this.task.setAttribute("sourceIP",sourceIP);
 this.task.setAttribute("destIP",destIP);

 // 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);
 // create a contact for the Grid resource
 Contact contact = new Contact(``myGridNode");
   .....
}
```

Figure 6.9: Java Code for Requesting a Network Resource

6.2.2 – Experimental Results

The effectiveness of network resource reservations, based on the integration of BB_{Basic} and G-QoSm, was evaluated on a local network test-bed. This section discusses the experiments and presents the corresponding validation results.

6.2.2.1 Network Test-bed

Figures 6.10 and 6.11 show the network test-bed, a *local area network* (LAN) of computing nodes and routing elements, with computing nodes representing the source and sink points – i.e. traffic senders and receivers. The routing elements use the Linux *iproute2* package to provide DiffServ capability to a Linux-based machine, and the Linux machine then acts as a PEP entity. Figure 6.10 shows the intra-domain architecture, while Figure 6.11 shows an inter-domain architecture.

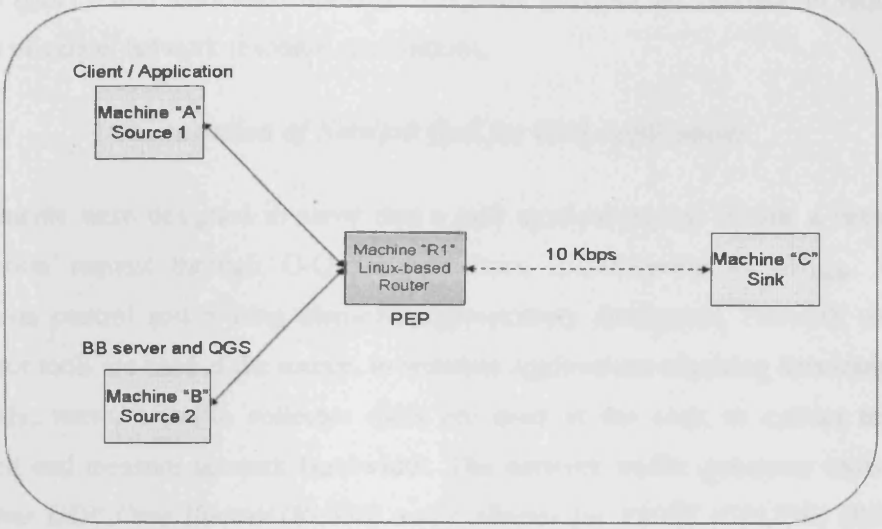

Figure 6.10: Network Setup for Intra-domain Architecture

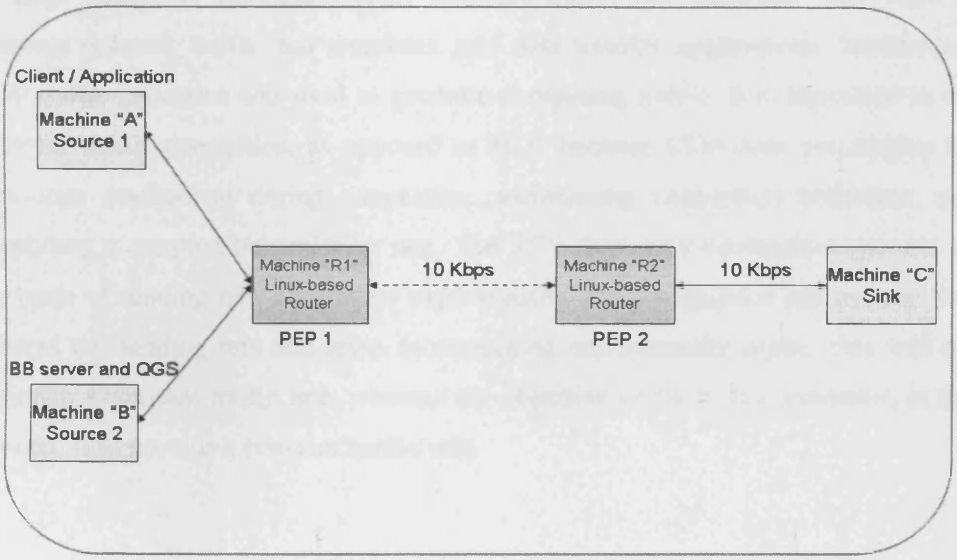

Figure 6.11: Network Setup for Inter-domain Architecture

BB_{Basic} comes with three separate modules: a BB server, to be installed in each administrative domain, to act as a PDP; a PEP module to be installed in each Linux routing element; and a MySQL database to be populated with the relevant data describing the network. For example, the database describes network topology, link capacities and the pre-defined SLAs with their service types – expedited forwarding

or *best effort* – and associated domains. G-QoSm provides an interface to request, modify or cancel network resource reservations.

6.2.2.2 *Demonstration of Network QoS for Grid Applications*

Experiments were designed to show that a grid application can initiate a network reservation request through G-QoSm and have it forwarded to BB_{Basic}, with admission control and routing elements appropriately configured. Network traffic generator tools are used at the source, to simulate applications requiring data transfer. Similarly, network traffic collector tools are used at the sink, to collect traffic received and measure network bandwidth. The network traffic generator tools are *Real-time UDP Data Emitter* (RUDE) and *Collector for RUDE* (CRUDE) (RUDE and CRUDE, 2004).

A User Datagram Protocol (UDP) constant-traffic-rate generator was used to generate network traffic that simulates grid data-transfer applications. Similarly, a UDP traffic generator was used to generate competing traffic. It is important to use UDP for traffic congestion, as opposed to TCP, because UDP does not employ the slow-start mechanism during congestion, maintaining congestion behaviour, and supporting a constant transmission rate. The TCP slow-start mechanism operates on the basis of sending rate increments exponentially until congestion occurs, and then reduces the sending rate and starts incrementing exponentially again. This will not maintain a constant traffic rate, whereas the objective of the traffic generator, in this context, is to provide a constant traffic rate.

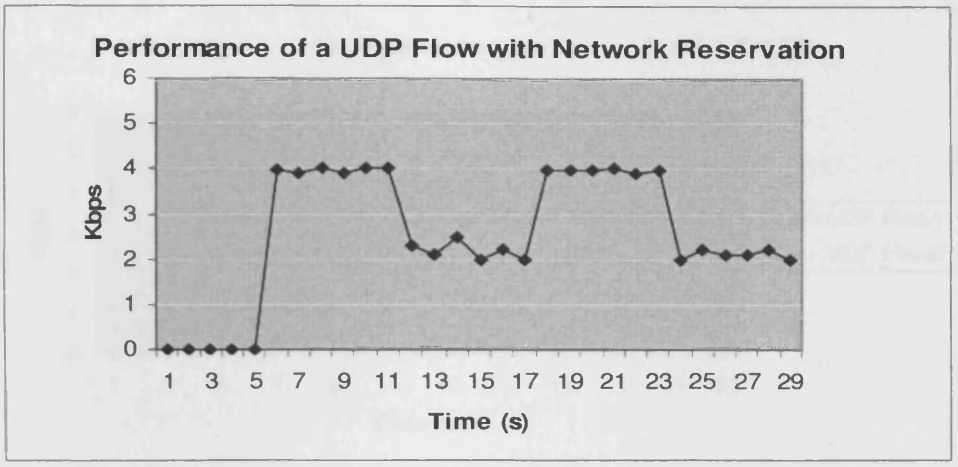

Figure 6.12: Network QoS under Congestion

Figure 6.12 shows the performance of network QoS for UDP traffic simulating a grid application under different situations. This experiment was conducted in the intra-domain architecture shown in Figure 6.10. The link between the router element and the sink was configured for a 10 Kbps stream, to easily congest the link. The UDP traffic under consideration was maintained from time t_5 to t_{29}. From t_5 to t_{10} the UDP traffic was sent without reservation – i.e. *best effort* – at 4 Kbps on an unloaded communication link from source to sink. From t_{11} to t_{16}, with the UDP flow still transmitting at 4 Kbps, random competing traffic was started to generate congestion; observations show that the UDP traffic could not maintain the 4 Kbps rate due to congestion. A network QoS reservation, for 4 Kbps, was made from t_{17} to t_{23} for the UDP traffic, with the competing traffic still generating congestion.

The result of the QoS reservation was that the UDP traffic managed to maintain the promised reservation rate, even though congestion was still operating. Finally, from t_{24} to t_{29} the reservation ended and the UDP traffic was unable to keep its 4 Kbps.

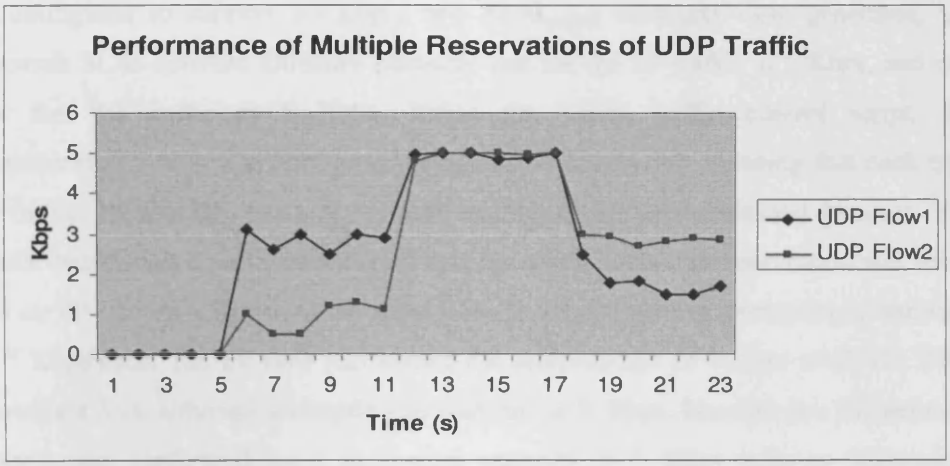

Figure 6.13: Multiple Network QoS Flows under Congestion

Figure 6.13 demonstrates multiple network QoS reservations under congestion. This is similar to the previous setup, with the link between the router and the sink configured to 10 Kbps. In this case, two UDP flows were generated. From t_5 to t_{10} the two UDP flows transmitted simultaneously at 5 Kbps, while the congestion continued. Reservations were established from t_{11} to t_{16} and the 2 flows maintained the promised resources. The DiffServ forwarding mechanism, at the routing element, is thus undertaking the correct traffic forwarding.

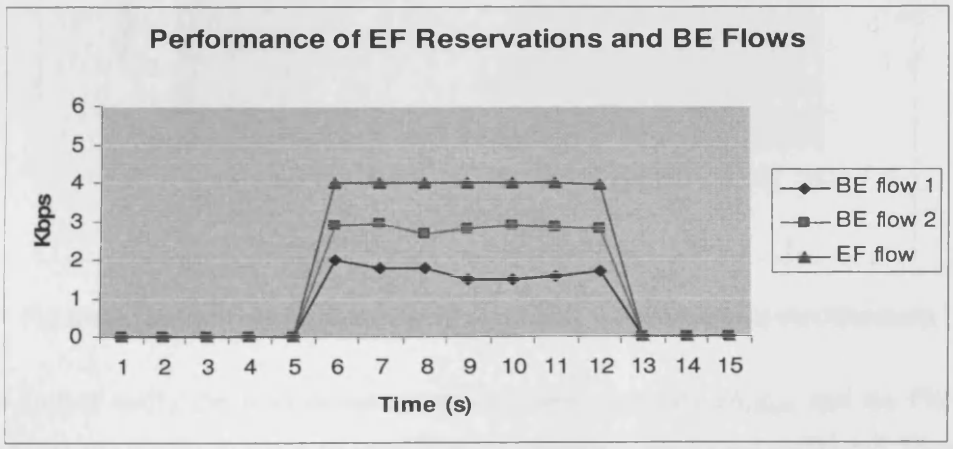

Figure 6.14: *Guaranteed* and *Best Effort* Network QoS

Figure 6.14 shows performance results for transmitting multiple traffic flows belonging to 2 different classes: EF – which can be mapped to *guaranteed* service in G-QoSm; and *best effort* (BE). The network link from the routing element to the sink

is configured to support 10 Kbps; two $SLA_{network}$ contracts were generated, i.e. network SLAs between DiffServ domains, one for the EF traffic at 5 Kbps, and one for the BE traffic at 5 Kbps. Using the Linux traffic control script, the communication link was configured to not allow *borrowing*, meaning that each type of traffic, EF and BE, must stay within the boundaries of the defined resource. The traffic performance was realised from t_5 to t_{13}, when a network reservation was made for an EF flow of 4 Kbps. At the same time, 2 BE flows were attempting to transmit at 5 Kbps each. The EF flow maintained the reserved rate of 4 Kbps while the 2 BE flows are less, although attempting to transmit at 5 Kbps, because the BE network source was configured for a maximum capacity of 5 Kbps with no borrowing. Therefore, the routing element shaped, and policed, the two BE flows to fit within the configured BE network resource. The concept of borrowing network resources is consistent with the adaptation model outlined in Algorithm 3.2. One can map the adaptation model into the network resource and use the borrow concept to implement the adaptive capacity of the adaptation model.

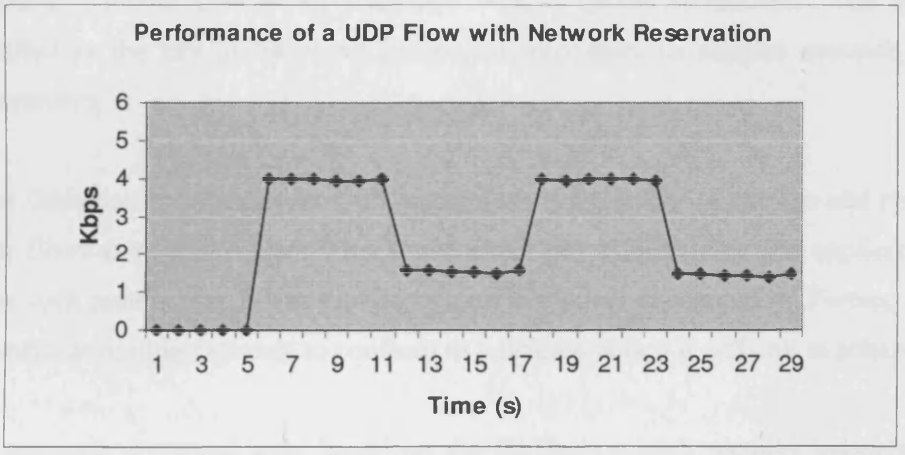

Figure 6.15: Network QoS under Congestion – Inter-**domain Architecture**

To further verify the inter-domain communication between BB_{Basic} and the PEPs, experiments similar to those on intra-domain communication were conducted. Figure 6.15 shows the results, which are similar to the intra-domain case, implying that BB_{Basic} is able to configure local PEPs as well as remote PEPs. The concept of inter-domain communication can be replicated, over a large number of administrative domains, making the proposed architecture scalable.

6.3 Summary

A G-QoSm prototype is used in a nanoscale application, as an illustrative example, to validate the usefulness of the proposed approach for the compute QoS in scientific applications. The architecture includes a set of components that abstract the use of QoS for the non-programmer. It is emphasised 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 utilise grid computing resources.

It is shown in this Chapter how compute QoS support at the middleware level provides a better application performance. This Chapter also focuses on evaluating the combination of G-QoSm and a BB, using a network established with Linux-based routers.

The provision of network QoS to support grid applications is presented, based essentially on the IETF DiffServ model. The DiffServ model is shown to provide acceptable network QoS when integrated with G-QoSm architecture. The BB is identified as the key architectural component necessary to support network QoS management.

A key limitation in any network QoS mechanism is the ability to manage and control traffic flows at internal routers. This is especially true in deploying grid applications, where such routers may not be owned by one individual or institution. Forcing such intermediate routing elements to conform to a defined policy is difficult to achieve.

The approach presented here, based on the DiffServ model, requires intermediate routers to adopt the DiffServ-expedited forwarding model. Consequently, the approach is restricted to routers that support this model – providing a traditional *best effort* service at other routers. The author knows of no other QoS-related work that avoids this need to manage intermediate routers.

Chapter 7 ~ Conclusions summarises questions addressed by this research, discusses contributions made and provides recommendations for further research.

Chapter 7 ~ Conclusion

7.1 Synopsis

This thesis proposes a *quality-of-service* (QoS) management system. QoS management is essential to provide guaranteed resource allocations with specified quality levels, and is a means to negotiate and establish *service level agreements* (SLAs), and then deliver services according to SLA specifications. A summary of the research findings, contributions and recommendations for future work is presented in this Chapter.

In Chapter 1 it is hypothesised that QoS management in a *service-oriented architecture* (SOA) can provide a guaranteed, reliable and consistent service-execution mechanism. Questions considered include:

- How can a QoS management system be presented as a Web Service (WS), in the context of SOAs, where users and applications interact through standard WS protocols?
- How can a typical service-oriented application utilise and benefit from use of such a QoS management approach?
- What performance gains can be obtained by an application using such a QoS management system in a SOA?

To answer these questions an abstract model for QoS management in SOAs was developed, aimed at maximising resource utilisation, while maintaining contracted SLAs. Maximising resource utilisation admits more SLA users to the system, which is possible with the flexible *range-based* SLA feature. The abstract model shows that the QoS problem – to determine, given multiple client requests, the optimal resource allocation to maximise utilisation and maintain requested QoS levels – is an optimisation problem.

To validate the model, the G-QoSm prototype was designed and built as a grid service in the context of grid computing. G-QoSm is modular in design, giving it flexibility to include new resource managers to support different resources, as, and when, they become available. Integrating new resource managers is possible because

of the uniform treatment of a variety of resource managers through a resource-specific interface layer. The architecture is a self-contained QoS management system which can be used with the Java CoG Kit client library. Consequently, a grid application that uses the Java CoG Kit has a natural transition into G-QoSm, and a new grid application can easily become QoS-aware.

This prototype was integrated with a scientific application of nanoscale structures, and used to evaluate computational QoS property. The network QoS property is evaluated through a simulation of grid data-transfer application. The evaluation is aimed at comparing the performance of the G-QoSm prototype to a standard grid middleware system without QoS management support, based on two measures:

- For computation QoS, the time taken to complete a process with QoS constraints, despite workloads generated by other applications utilising system resources.
- For network QoS, the ability of an application with QoS constraints to maintain a promised rate of data transfer while other applications are utilising system resources.

Performance results and analysis, based on the G-QoSm approach, demonstrate the usefulness of a QoS management approach in SOAs, and, in particular, in grid computing. The results show that in the case of computational QoS support, the performance of the application with QoS support yielded improved performance and provided reliable and consistent application execution. In this context, reliable implies that when an application is given a SLA indicating certain resources will be available, at certain pre-defined time, with the expected performance levels, then this is an assurance the application will find these resources available when the time comes. Similarly, consistent implies that the application will receive the expected performance throughout the SLA validity period. The results also show that the introduction of QoS generates some processing overhead – this overhead is, however, small, and negligible when compared to the overhead generated by WS protocols, especially when invoking services using the SOAP protocol. The overhead generated by the QoS management system is in the order of 100 ms per request. Essentially this overhead results from the negotiation process during the establishment phase of the QoS session, and the 100 ms overhead constitutes 100% of the negotiation

overhead per request, with request, in this context, meaning a single request, from the client, and a corresponding reply from the QoS management entity.

Similarly, in the case of network performance, results show that the simulated application can successfully maintain the promised rate of data transfer, while other applications utilise network resources, throughout the SLA validity period. The provision of network QoS to support grid applications is based essentially on the IETF DiffServ model. The Bandwidth Broker (BB) is identified as the key architectural component necessary to support network QoS management. A key limitation with network QoS approaches is the ability to manage and control networking elements to conform to a defined policy. This is especially true when deploying grid applications where such networking elements may not be owned by one individual or institution.

QoS abstractions are also presented for building QoS-based applications in the context of service-oriented grids. These abstractions, presented as an *application programming interface* (API), will assist application developers in building QoS-aware grid applications.

G-QoSm is not limited to service-oriented grids, and is also suitable for applications in other SOAs, and the G-QoSm model can, for example, be applied in peer-to-peer computing (Rana *et al.* 2005).

7.2 Contributions

A new abstract model for resource management, based on QoS for service-oriented architectures is presented. This model is a general type for QoS management in SOAs and can be applied in various architectures. Although this model is designed for SOAs, the concepts developed in the model are not restricted to SOAs. The key advantages of SOAs are loose coupling, in application-to-application interaction or application to data sources, and inter-operability support.

A novel protocol for agreement-based QoS negotiation, establishing a SLA as a contract between service consumer and provider, is developed. This protocol is

particularly useful when designing QoS brokers for a distributed computing environment.

A new approach to resource selection, based on QoS properties, is presented. This is possible through the extension of a standard registry system, such as UDDI. The extension enables the registry system to support service publishing and discovery, based on QoS properties as outlined in Chapter 4. A service selection approach is introduced to select the best match based on a client's application requirements.

Two mechanisms for resource allocation (i) *time domain* and (ii) *resource domain* are presented. Time domain is suitable for applications requiring high-performance computing resources, while resource domain is suitable for small applications and services requiring relatively limited resources with QoS guarantees.

A new technique for advance resource reservation in grids, for single, or multiple, resources is developed. Most reservation systems deal with only one type of resource per request, as in GARA; however, in grid systems applications are normally interested in using multiple resources simultaneously. The proposed technique for reserving multiple resources, both computational and network is effective for grid applications.

Resources can become congested or even fail, leading to QoS degradation, and require adaptation mechanisms to maintain SLA compliance. Adaptation mechanisms are developed to compensate for such QoS degradation and to optimise resource utilisation, as discussed in Chapter 3. The adaptation approach is based on reserving extra resources for the *guaranteed* class of service.

In summary, the main contribution of this work is an approach to enhance the basic principles of the SOA in supporting QoS, which enables the execution of applications with resource QoS guarantees, based on pre-established agreements. This QoS support is realised by introducing a QoS management component in middleware systems.

7.3 Further Research

Various issues arise which present opportunities for future research in this field.

7.3.1 – Cost Model

A cost model to price resources would improve the G-QoSm model. The need for such a model becomes clear when considering multiple applications competing simultaneously for immediate or advance reservations of a finite set of resources. With a cost model, a QoS management system would be able to limit competition while still generating necessary revenue, which can be realised by applying a cost-related reservation strategy, such as increasing the cost when resources become limited. Such a cost model could be derived from business and economic theories.

7.3.2 – Reservation Strategies

A resource reservation strategy is a key function in QoS management systems, and introducing advanced strategies, or approaches, for resource reservation can improve resource utilisation. Reservation strategies, based on statistical information for applications and resources, can be utilised to achieve this; such statistical information can be application-profiling data, application usage patterns or the use of probability functions (Rolia *et al.* 2003).

7.3.3 – QoS for Workflow and Task Graphs

In this context, a task represents a unit of execution on a grid or *job*. Certain applications require a more sophisticated execution framework facilitating complex execution patterns and dependencies. A *task graph* – a directed acyclic graph – for execution control flows between multiple tasks can be modeled (Amin *et al.* 2004). A *task graph handler* enforces execution ordering on the task graph. QoS support, as available in G-QoSm, can be integrated with the task graph handler to support execution with QoS properties.

Similarly, in the context of workflow management, G-QoSm could be integrated with a workflow scheduling engine to form QoS-enabled workflow applications, enabling execution of such workflow applications on resources with QoS provisions.

7.3.4 – Monitoring Service

Monitoring resource utilisation is an important QoS management function during the active phase of a QoS session – useful for accounting, adaptation and resource profiling. An investigation into the design of a monitoring service, to provide feedback on resource utilisation to the QoS manager, would be useful. For example; the Grid Resource Monitoring (GridRM) project (Baker and Smith, 2003) and the Network Weather Service (NWS) project (Wolski, Spring and Hayes, 1999) can provide such a monitoring functionality.

The monitoring service can report on resource utilisation during the active phase of the QoS session. This service can be linked with the *allocation manager* and the *reservation manager* of the G-QoSm for SLA compliance verification and adaptation purposes.

7.3.5 – Prediction Service

In the G-QoSm architecture, an application can request services from the QoS Manager, even though the QoS Manager has no QoS information about the requested service. Here the QoS manager consults the registry service for resource and QoS specifications, suggested as sufficient to run the service.

It would benefit the G-QoSm to have a prediction method for determining resource and QoS specifications for a requested service, in the environment in which the service is to be executed. Such a service could reduce over, or under, reservation and provide for *just sufficient* resource reservation, as in the reservation technique in Chu and Nahrstedt (1999) in Chapter 2. The *PACE* project, at Warwick University, is a prediction service which may well deliver these services within G-QoSm (Jarvis *et al.* 2003). Systems with prediction capabilities, such as PACE, can be used in the G-QoSm architecture to provide QoS information related to services, which information, and, in particular, the service profile, can then be published in the registry service.

Bibliography

Al-Ali, R; Rana, O; Walker, D; Jha, S and Sohail, S. (2002a). G-QoSm: Grid Service Discovery using QoS Properties. *Computing and Informatics Journal*, 21(4):363-382, 2002. Slovak Academic Press Ltd.

Al-Ali, R; Rana, O; Walker, D; Jha, S and Sohail, S. (2002b). Grid Service Discovery Based on Quality of Service Characteristics. In *Proceedings of the e-Science AHM02 Proceedings*, 2002.

Al-Ali, R; Amin, K; von Laszewski, G; Rana, O and Walker, D. (2003a). An OGSA-Based Quality of Service Framework. *The Second International Workshop on Grid and Cooperative Computing (GCC2003)*, Shanghai, China, December 2003. Springer Verlag.

Al-Ali, R; Hafid, A; Rana, O and Walker, D. (2003b). QoS Adaptation in Service-Oriented Grids. *Proceedings of the 1st International Workshop on Middleware for Grid Computing (MGC2003) at ACM/IFIP/USENIX Middleware 2003*. Rio de Janeiro, Brazil, June 2003. ISBN 85-87926-03-9.

Al-Ali, R; Rana, O and Walker, D. (2003c). G-QoSm: A Framework for Quality of Service. In *Proceedings of the e-Science AHM03*, Nottingham, UK, 2003.

Al-Ali, R; ShaikhAli, A; Rana, O and Walker, D. (2003d). Supporting QoS-Based Discovery in Service-Oriented Grids. In *Proceedings of IEEE Heterogeneous Computing Workshop (HCW'03)*, Nice, France, 2003. IEEE Computer Society Press.

Al-Ali, R; Amin, K; von Laszewski, G; Hategan, M; Rana, O; Walker, D and Zaluzec, N. (2004a). QoS Support for High-Performance Scientific Applications. *Proceedings of the IEEE/ACM 4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*. Chicago IL, USA, April 2004. IEEE Computer Society Press.

Al-Ali, R; Amin, K; von Laszewski, G; Rana, O; Walker, D; Hategan, M and Zaluzec, N. (2004b). Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, 2(2): 163-182, 2004. Kluwer Academic Publishers.

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

Al-Ali, R; Rana, O; von Laszewski, G; Hafid, A; Amin, K and Walker, D. (2005) A Model for Quality-of-Service Provision in Service Oriented Architectures. *International Journal of Grid and Utility Computing*, 2005.

Al-Ali, R; Sohail, S; Rana, O; Hafid, A; von Laszewski, G; Amin, K; Jha, S and Walker, D. (2004d) Network QoS Provision for Distributed Grid Applications. *International Journal of Simulations Systems, Science & Technology*, 5(5), December 2004.

Akram, A and Rana, O. (2003a). Structuring Peer-2-Peer Communities, *3rd IEEE International Conference on Peer-2-Peer Computing*, Linkoping, Sweden, September 2003. IEEE Computer Society Press.

Akram, A and Rana, O. (2003b). Organising Service-Oriented Peer Collaborations, *International Conference on Service Oriented Computing*, Italy, December 2003. Springer Verlag.

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

Amin, K; von Laszewski, G; Al-Ali, R; Rana, O and Walker, D. (2005). *An Abstraction Model for a Grid Execution Service.* Journal of Systems Architecture, 2005.

Andrieux, A; Czajkowski, K; Dan, A; Keahey, K; Ludwig, H; Pruyne, J; Rofrano, J; Tuecke, S and Xu, M. (2004). Web Services Agreement Specification (WS-Agreement). Global Grid Forum, GRAAP-WG Author Contributions, May 2004.

Argonne National Laboratory. (2004). The Globus Alliance.
Web Site: http://www.globus.org/. Last visited: February 2004.

Aurrecoechea, C; Campbell, A and Hauw, L. (1995). A Survey of Quality of Service Architectures. Technical Report MPG-95-18, Lancaster University, 1995.

Baker, M; Buyya, R and Laforenza, D. (2000). The Grid: International Efforts in Global Computing. *International Conference on Advances in Infrastructure for Electronic Business, Science and Education on the Internet (SSGRR 2000)*, Rome, Italy, 2000. ISBN 88-85280-52-8.

Baker, M and Smith, G. (2003). GridRM: An Extensible Resource Management System. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, 2003. IEEE Computer Society Press, ISBN 0-7695-2066-9.

Barden, R; Clark, D and Shenker, S. (1994). *Integrated Services in the Internet Architecture: an Overview.* Internet request for Comments RFC1633, IETF, June 1994.

Bent, J; Venkataramani, V; LeRoy N; Roy A; Stanley, J; Arpaci-Dusseau, A; Arpaci-Dusseau R. H., and Livny, M. (2002). "Flexibility, Manageability, and Performance in a Grid Storage Appliance", *Proceedings of the Eleventh IEEE*

Symposium on High Performance Distributed Computing, Edinburgh, Scotland, July 2002. IEEE Computer Society Press.

Bhatti, S; Sørensen, S; Clarke, P and Crowcroft, J. (2003). Network QoS for Grid Systems. *International Journal of High Performance Computing Applications, Special Issue on Grid Computing: Infrastructure and Applications*, 17(3), 2003.

Bhoj, P; Singhal, S and Chutani, S. (1998). SLA Management in Federated Environments. Technical Report HPL-98-203, Hewlett-Packard Company, December 1998.

Blake, S; Blake, D; Carlson, M; Davies, E; Wang, Z and Weiss, W. (1998). An Architecture for Differentiated Service. Internet RFC 2475, 1998.

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

Burchard, L; Hovestadt, M; Kao, O; Keller, A and Linnert, B. (2004). The Virtual Resource Manager: An Architecture for SLA-aware Resource Management. In *Proceedings of IEEE CCGrid'04*, Chicago, US, 2004. IEEE Computer Society Press.

Campbell, A; Coulson, G; Garcia, F and Hutchison, D. (1993). Resource Management in Multimedia Communication Stacks. In *Proc. 4th IEEE Conference on Telecommunications, Manchester, UK*, pages 287-295, 1993. IEEE Computer Society Press.

Cardei, I; Jha, R; Cardei, M and Pavan, A. (2000). Hierarchical Architecture for Real-Time Adaptive Resource Management. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 415-434, New York, USA, 2000. Springer-Verlag, ISBN:3-540-67352-0.

Chandra, P; Fisher, A; Kosak, C; Eugene Ng, T.S; Steenkiste, P; Takahashi, E and Zhang, H. (1998). Darwin: Customizable Resource Management for Value-Added Network Services. In *Sixth International Conference on Network Protocols*, pages 177-188, Austin, October 1998. IEEE Computer Society Press.

Chan, K; Sahita, R; Hahn, S and McCloghrie, K. (2003). *Differentiated services quality of service policy information base* . Internet request for comments RFC3317, IETF, March 2003.

Chan, K; Seligson, J; Durham, D; Gai, S; McCloghrie, K; Herzog, S; Reichmeyer, F; Yavatkar, R and Smith, A. (2001). *Cops usage of policy provisioning (COPS-PR)*. Internet request for comments RFC3084, IETF, Mar 2001.

Chu, H and Nahrstedt, K. (1999). CPU Service Classes for Multimedia Applications. In *IEEE International Conference on Multimedia Computing and Systems '99*, Florence, Italy, 1999. IEEE Computer Society Press.

Czajkowski, K; Foster, I; Karonis, N; Kesselman, C; Smith, M and Tueck, S. (1998). "A Resource Management Architecture for Metacomputing Systems." *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,* 62-82, 1998. Springer-Verlag, ISBN:3-540-64825-9.

Czajkowski, K; Fitzgerald, S; Foster, I and Kesselman, C. (2001). Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE High Performance Distributed Computing*, pages 181-184, 2001. IEEE Computer Society Press.

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

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

Czajkowski, K; Ferguson, D; Foster, I; Frey, J; Graham, S; Sedukhin, I; Snelling, D; Tuecke, S and Vambenepe, W. (2004). The WS-Resource Framework (Version 1.0). The Globus Alliance, Web Site: http://www.globus.org/wsrf/ Last visited: May 2004.

Czerwinski, S; Zhao, B; Hodes, T; Joseph, A and Katz, R. (1999). An Architecture for a Secure Service Discovery Service. In *Mobile Computing and Networking*, pages 24-35, 1999. ACM Press.

Davis, R. and Smith R.G. (1983). 'Negotiation as a Metaphor for Distributed Problem Solving'. Artificial Intelligence 20, 63-109, 1983. Elsevier Science Publishers.

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

Dialani, V; Miles, S; Papay, J and Moreau, L. (2002). The Architecture of UDDI-M, 2002. Technical report, University of Southampton UK.

Durham, D; Boyle, J; Cohen, R; Herzog, S; Rajan, R and Sastry, A. (2000). *The COPS (common open policy service) protocol.* Internet request for comments RFC2748, IETF, Jan 2000.

Fitzgerald, S; Foster, I; Kesselman, C; von Laszewski, G; Smith, W and Tuecke, S. (1997). A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the IEEE 6th Symposium on High-Performance Distributed Computing*, pages 365-375, 1997. IEEE Computer Society Press. ISBN:0-8186-8117-9.

Foster, I; Kesselman, C; Lee, C; Lindell, R; Nahrstedt, K and Roy, A. (1999). A Distributed Resource Management Architecture that Supports Advance Reservation and Co-Allocation. In *Proceedings of the IEEE/IFIP International Workshop on Quality of Service (IWQOS'99)*, pages 27-36, London, UK, 1999. IEEE Computer Society Press.

Foster, I; Roy, A and Sander, V. (2000). A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service (IWQOS)*, pages 181-188, Pittsburgh, PA, June 2000. IEEE Computer Society Press.

Foster, I; Kesselman, C and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001. Lecture Notes in Computer Science.

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

Graham, S; Simeonov, S; Boubez, T; Davis, D; Daniels, G; Nakamura, Y and Neyama, R. (2002). Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, 2002. SAMS Publishing. ISBN 0-672-32181-5.

GRIA Project. (2004). GRIA: Grid Resources for Industrial Applications. Web Site: http://www.gria.org/ Last visited: May 2004.

Gunter, D; Tierney, B; Jackson, K; Lee, J and Stoufer, M. (2002). "Dynamic Monitoring of High-Performance Distributed Applications." *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing, (HPDC-11)*, July, 2002. IEEE Computer Society Press.

Haas, R; Droz, P and Stiller, B. (2001). A hierarchical Mechanism for the Scalable Deployment of Services over Large Programmable and Heterogeneous Networks. In *Proceedings of ICC2001 The IEEE International Conference on Communications*, 2001. IEEE Computer Society Press.

Hafid, A; Bochmann, G and Kerherve, B. (1996). A Quality of Service Negotiation Procedure for Distributed Multimedia Presentational Applications. In *HPDC '96*, pages 330-339, Syracuse, NY, USA, 1996. IEEE Computer Society Press.

Hafid, A and Bochmann, G. (1998). Quality of Service Adaptation in Distributed Multimedia Applications. *ACM Multimedia Systems Journal*, 6(5):299-315, 1998. Springer-Verlag.

Hafid, A; Bochmann, G and Dssouli, R. (1998). A Quality of Service Negotiation Approach with Future Reservation (NAFUR): A Detailed Study. *Computer Networks and ISDN*, 30(8), 1998. Elsevier Science, ISSN 0169-7552.

Halim, H and Darmadi, M. (2000). Implementation of Bandwidth Broker using COPS-PR. *Honours thesis report, School of Computer Science and Engineering*, UNSW, Nov 2000.

Henricksen, K and Indulska, J. (2001). Adapting the Web Interface: An Adaptive Web Browser. In *Second Australasian User Interface Conference (AUIC'01)*, Gold Coast, Queensland, Australia, 2001. IEEE Computer Society Press.

Jacobson, V; Nichols, K and Poduri, K. (1999). An Expedited Forwarding PHB. Internet RFC 2598, IETF, 1999.

Jarvis, S; Spooner, D; Keung, H; Dyson, J; Zhao, L and Nudd, G. (2003). Performance-based Middleware Services for Grid Computing. In *Fifth International Workshop on Active Middleware Services (AMS 2003), held as part of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, Seattle, USA, 2003. IEEE Computer Society Press.

Karsten, M; Berier, N; Wolf, L and Steinmetz, R. (1999). A Policy-Based Service Specification for Resource Reservation in Advance. In *International Conference on Computer Communications (ICCC'99)*, 1999. ISBN 1-891365-05-3.

Keahey, K and Motawi, K. (2003). The Taming of the Grid: Virtual Application Service. Argonne National Laboratory, Technical Memorandum, 2003.

Kim, K and Nahrstedt, K. (2000). A Resource Broker Model with Integrated Reservation Scheme. In *IEEE International Conference on Multimedia and Expo (ICME2000)*, 2000. IEEE Computer Society Press.

Lefevre, L; Pham, C; Primet, P; Tourancheau, B; Gaidioz, B; Gelas, J and Maimour, M. (2001). Active Networking Support for the Grid. In *IFIP-TC6 Third International Working Conference on Active Networks, IWAN'01*, 2001. Springer-Verlag, ISBN:3-540-42678-7.

Lican, H; Zhaohui, W and Yunhe, P. (2003). A Scalable and Effective Architecture for Grid Services Discovery. In *Proceedings of SemPGRID'031st Workshop on Semantics in Peer-to-Peer and Grid Computing*, Budapest, Hungary, 2003.

Lim, K and Stadler, R. (2001). A Navigation Pattern for Scalable Internet Management. In *Proceedings IFIP/IEEE Intl Symposium on Integrated Network Management*, 2001. Seattle, WA, USA. IEEE Computer Society Press.

Liming, L. (2004) The Globus Toolkit Ecosystem. *Argonne National Laboratory - The Globus Alliance, GRIDS Center/NFS Middleware Initiative. Slide presentation at conference: All-Hands-Meeting*, 2004. Nottingham, UK.

Liu, Z; Squillante, M and Wolf, J. (2001). On Maximizing Service-Level-Agreement Profits. *Proceedings of the Third ACM Conference on Electronic Commerce*, 213-223, 2001. ACM Press, ISSN:0163-5999.

Lourenco, H and Serra, D. (2002) Adaptive Search Heuristics for the Generalized Assignment Problem. *Mathware and Soft Computing*, 9(2-3): 209-234, 2002. European Society for Fuzzy Logic and Technology (EUSFLAT).

Ludwig, S and van Santen, P. (2002). A Grid Service Discovery Matchmaker based on Ontology Description. In *Proceedings of 2nd International EuroWeb2002 Conference*, Oxford, UK, 2002. British Computer Society.

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

Madja, E; Hafid, A; Dssouli, R; Bochmann, G and Gecsei, J. (1998). Meta-data Modelling for Quality of Service Management in the World Wide Web. In *Proc. of Int. Conf. on Multimedia Modeling*, 1998. IEEE Computer Society Press.

Moreau, L; Dialani, V; Miles, S and Liu, X. (2002). Architectural Issues in myGrid. In *UK e-Science All-Hands Meeting*, 2002.

Nahrstedt, K and Smith, J. (1995) The QoS Broker. *IEEE Multimedia*, 2(1):53-67 1995. IEEE Computer Society Press.

National Fusion Collaboratory. The (2005). Web Site: http://www.fusiongrid.org Last visited: September 2005.

NetSolve/GridSolve Project (2004) http://icl.cs.utk.edu/netsolve/ Last visited: August 2004.

Nguyen, T; Boukhatem, N; Doudane, Y and Pujolle, G. (2002). COPS-SLS: A Service Level Negotiation Protocol for the Internet. *IEEE Communications Magazine*, 40(5), 2002. IEEE Computer Society Press.

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

Pard, J; Baek, J and Hong, J. (2001). Management of Service Level Agreements for Multimedia Internet Service Using a Utility Model. *IEEE Communications Magazine*, 39(5), 2001. IEEE Computer Society Press.

Parunak, H.V.D. (1987). 'Distributed Artificial Intelligence', Chapter: Manufacturing Experience With the Contract Net, pp. 285-310, Research Notes in Artificial Intelligence, Los Altos, CA, 1987. Morgan Kaufmann Publishers.

Pham, K and Nguyen, R. (2003) *Implementation of Bandwidth Broker in Java.* Undergraduate thesis report, School of Electrical Engineering and Telecommunications, UNSW, Jun 2003.

QBone Signaling Design Team. (2002). Final Report: http://qos.internet2.edu/wg/documentsinformational/20020709-chimento-etal-qbonesignaling/

Rana, O; Akram, A; Al-Ali, R; Walker, D; von Laszewski, G and Amin, K. (2005). *Quality of Service Based Grid Communities.* Chapter in Book "*Web Services and Agent-Based Engineering*", 2005, Springer Verlag.

Rana, O; Bunford-Jones, D; Walker, D; Addis, M; Surridge, M and Hawick, K. (2001). Resource Discovery for Dynamic Clusters in Computational Grids. In *Proceedings of the Heterogeneous Computing Workshop at IPDPS/SPDS,* San Francisco, USA, 2001. IEEE Computer Society Press.

Rana, O; Winikoff, M; Padgham, L and Harland, J. (2002). Applying Conflict Management Strategies in BDI Agents for Resource Management in Computational Grids. *Twenty-Fifth Australasian Computer Science Conference (ACSC2002),* Monash University, Melbourne, Victoria, Australia, 2002. Australian Computer Society, ISBN 0-909-92582-8.

RAP, (2000). *Resource allocation protocol (rap)* At http://www.ietf.org/charters/ manet-charter.html 2000.

Rolia, J; Pruyne, J; Zhu, X and Arlitt, M. (2003). Grids for Enterprise Applications. *Proceedings of the 9 workshop on Job Scheduling Strategies for Parallel Programs.* Seattle, June 2003. Lecture Notes in Computer Science.

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

RUDE & CRUDE, (2004). Web Site: http://rude.sourceforge.net/ July 2004.

Sahai, A; Ouyang, J; Machiraju, V and Wurster, K. (2001). Specifying and Guaranteeing Quality of Service for Web Services through Real Time Measurement and Adaptive Control. Technical Report HPL-2001-134, E-Services Software Research Department. HP Labs, Palo-Alto, CA, USA, 2001.

Sahai, A; Graupner, S; Machiraju, V and van Moorsel, A. (2003). Specifying and Monitoring Guarantees in Commercial Grids through SLA. In *Proceedings of the 3rd IEEE/ACM CCGrid'03,* Hong Kong, 2003. IEEE Computer Society Press.

ShaikhAli, A; Rana, O; Al-Ali, R and Walker, D. (2003). *UDDIe: An Extended Registry for Web Services.* Proceedings of the Service Oriented Computing: Models, Architectures and Applications, SAINT-2003, Orlando, Florida, USA, January 2003. IEEE Computer Society Press.

Shenker, S; Partridge, C and Guerin, R. (1997). Specification of Guaranteed Quality of Service. *Internet Engineering Task Force, RFC 2212*, 1997.

Shi, Z; Yu, T and Liu, L. (2003). MG-QoS: QoS-Based Resource Discovery in Manufacturing Grid. In *Proceedings of the Second International Workshop on Grid and Cooperative Computing (GCC2003)*, Shanghai, China, 2003. Lecture Notes in Computer Science, Springer Verlag.

Sohail, S; Pham, K; Nguyen, R and Jha, S. (2003). Bandwidth Broker Implementation - Circa-Complete and Integrable. *Proceedings of 7th International Symposium on Digital Signal Processing and Communication Systems*, Coolangata, Australia, 2003. National ICT Australia Limited.

Taylor, I. J., (2005). *From P2P to Web Services and Grids*. Springer-Verlag, London Limited 2005. ISBN 1852338695.

Taylor, I; Shields, M; Wang, I and Rana, O. (2003). Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199-217. Kluwer Academic Press.

Teitelbaum, B; Hares, S; Dunn, L; Neilson, R; Vishy Narayan, R and Reichmeyer, F. (1999) Internet2 QBone: Building a testbed for differentiated services, *IEEE Network*, 13(5):8–16, September/October 1999. IEEE Computer Society Press.

'TeraGrid' Web Page, (2001). http://www.teragrid.org/
Last visited: July 2004.

'The Global Grid Forum' Web Page, (2004). http://www.gridforum.org
Last visited: August 2004.

The Java CoG Kit Project. (2004). http://www-unix.globus.org/cog
Last visited: September 2004.

The Legion Project. (2004). http://www.cs.virginia.edu/~legion/
Last visited: August 2004.

UDDI Project. (2004). UDDI: Universal Description, Discovery and Integration. Web Site: http://www.uddi.org Last visited: May 2004.

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

von Laszewski, G; Pieper, G and Wagstrom, P. (2003). Gestalt of the Grid. *in Performance Evaluation and Characterization of Parallel and Distributed Computing Tools. Series on Parallel and Distributed Computing*, 2004. Wiley.

Wolski, R; Spring, N and Hayes, J. (1999) The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, Volume 15, Numbers 5-6, pp. 757-768, October, 1999. Elsevier Science.

Wroclawski, J. (1997). Specification of the Controlled-load Network Element Service. *Internet Engineering Task Force, RFC 2211*, 1997.

Xiao, X and Ni, L. (1999). Internet QoS: A Big Picture. *IEEE Network*, 13(2):8-18, 1999. IEEE Computer Society Press.

Xu, D; Nahrstedt, K and Wichadakul, D. (2001). QoS-Aware Discovery of Wide-Area Distributed Services. In *Proceedings of the First IEEE/ACM Cluster Computing and the Grid (CCGrid'01)*, 2001. IEEE Computer Society Press.

Yemini, Y; Goldszmidt, G and Yemini, S. (1991). Network Management by Delegation. In *Proceedings of Intl. Symposium on Integrated Network Management*, 1991. Elsevier Science Publishers.

Zaluzec, N. (2004). Argonne National Laboratory TPM/AAEM Collaboratory. See Web Site at: http://tpm.amc.anl.gov/ Last visited: September 2004.

Appendix A

QGS Service WSDL Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://qos.cog.globus.org/QoS"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:gridservicesoapbinding="http://www.gridforum.org/namespaces/2003/03/OG
SI/bindings" xmlns:impl="http://qos.cog.globus.org/QoS"
xmlns:intf="http://qos.cog.globus.org/QoS"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><wsdl:import
location="../../ogsi/ogsi_bindings.wsdl"
namespace="http://www.gridforum.org/namespaces/2003/03/OGSI/bindings"/>
 <wsdl:types>
  <schema targetNamespace="http://qos.cog.globus.org/QoS"
       xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="serviceRequest">
    <complexType>
     <sequence>
      <element name="in0" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="serviceRequestResponse">
    <complexType>
     <sequence>
      <element name="serviceRequestReturn" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="confirmSlaOffer">
    <complexType>
     <sequence>
      <element name="in0" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="confirmSlaOfferResponse">
    <complexType>
     <sequence>
      <element name="confirmSlaOfferReturn" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="rejectSlaOffer">
    <complexType>
     <sequence>
      <element name="in0" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="rejectSlaOfferResponse">
    <complexType>
     <sequence>
      <element name="rejectSlaOfferReturn" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
```

```
 </schema>
</wsdl:types>

  <wsdl:message name="rejectSlaOfferRequest">
     <wsdl:part element="impl:rejectSlaOffer" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="confirmSlaOfferResponse">
     <wsdl:part element="impl:confirmSlaOfferResponse" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="serviceRequestResponse">
     <wsdl:part element="impl:serviceRequestResponse" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="serviceRequestRequest">
     <wsdl:part element="impl:serviceRequest" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="confirmSlaOfferRequest">
     <wsdl:part element="impl:confirmSlaOffer" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="rejectSlaOfferResponse">
     <wsdl:part element="impl:rejectSlaOfferResponse" name="parameters"/>
  </wsdl:message>
  <wsdl:portType name="QoSPortType">
     <wsdl:operation name="serviceRequest" parameterOrder="">
        <wsdl:input message="impl:serviceRequestRequest"
            name="serviceRequestRequest"/>
        <wsdl:output message="impl:serviceRequestResponse"
            name="serviceRequestResponse"/>
     </wsdl:operation>
     <wsdl:operation name="confirmSlaOffer" parameterOrder="">
        <wsdl:input message="impl:confirmSlaOfferRequest"
            name="confirmSlaOfferRequest"/>
        <wsdl:output message="impl:confirmSlaOfferResponse"
            name="confirmSlaOfferResponse"/>
     </wsdl:operation>
     <wsdl:operation name="rejectSlaOffer" parameterOrder="">
        <wsdl:input message="impl:rejectSlaOfferRequest"
            name="rejectSlaOfferRequest"/>
        <wsdl:output message="impl:rejectSlaOfferResponse"
            name="rejectSlaOfferResponse"/>
     </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="QoSServiceSoapBinding" type="impl:QoSPortType">
     <wsdlsoap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="serviceRequest">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="serviceRequestRequest">
           <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                  use="literal"/>
        </wsdl:input>
```

```
            <wsdl:output name="serviceRequestResponse">
               <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                      use="literal"/>
            </wsdl:output>
         </wsdl:operation>
         <wsdl:operation name="confirmSlaOffer">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="confirmSlaOfferRequest">
               <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                      use="literal"/>
            </wsdl:input>
            <wsdl:output name="confirmSlaOfferResponse">
               <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                      use="literal"/>
            </wsdl:output>
         </wsdl:operation>
         <wsdl:operation name="rejectSlaOffer">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="rejectSlaOfferRequest">
               <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                      use="literal"/>
            </wsdl:input>
            <wsdl:output name="rejectSlaOfferResponse">
               <wsdlsoap:body namespace="http://qos.cog.globus.org/QoS"
                      use="literal"/>
            </wsdl:output>
         </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="QoSService">
         <wsdl:port binding="impl:QoSServiceSoapBinding" name="QoSService">
            <wsdlsoap:address
                location="http://localhost/ogsa/services/QoSService"/>
         </wsdl:port>
      <wsdl:port binding="gridservicesoapbinding:GridServiceSOAPBinding"
                name="GridServiceSOAPBindingPort"><wsdlsoap:address
                location="http://localhost/ogsa/services/QoSService"/>
</wsdl:port>
</wsdl:service>
```

Appendix B

QGS Installation

QGS, part of G-QoSm, provides access to Grid resources with QoS guarantees. Two resource allocation strategies are supported: (a) time-domain, and (b) resource-domain.

- Time-domain entails the user having full access to the computer resource where the QGS is installed; and the user can submit job(s) to this particular resource throughout the period defined in the QoS agreement.
- Resource-domain is gaining access to specific computation capacity of the Grid node, for a period of time defined in the QoS guarantees.

Note: In the time-domain strategy the Grid resource is dedicated, while in the resource-domain strategy, the resource is shared

B.1. Installation Prerequisites

Ensure the following components are properly installed and configured:

1. Globus toolkit 3.0, or later versions - full installation or the 'core'
2. Java CoG kit 1.1a, or later versions from the Java CoG Kit project web site. The QoS package from http://users.cs.cf.ac.uk/Rashid/qos/ This QoS package should be placed in the downloaded CoG as a directory, under the directory /modules of the CoG as: ../modules/qos
3. Dynamic Soft Real-time scheduler (DSRT), available with this distribution – make sure you:
    - use the DSRT with this distribution as it has some customized API
    - Edit the file "config.txt" available in the root directory of this distribution, with the DSRT installation path and save the file in the '.globus/' directory.
4. Java VM and apache ant.

B.2. Compilation and Service Deployment

1. Edit the file 'build.properties' in the installation root directory with the right value of the 'ogsa.root', which should be set to the OGSA installation.

2. From the installation directory, run the convenient script, created by the GT3 team, as shown below, to compile the QGS service and create the appropriate jar and gar files.
   ./compileService.sh org/globus/cog/qos/imple/Qgs.java
   If all goes well, then you should have a build directory with all the jar, gar and compiled classes.

3. Create a proxy. If you don't have a valid one; one way to do this is from the CoG_dir/bin: enter the following: ./visual-proxy-init

4. From the OGSA installation directory deploy the service by entering the following command:
   ant deploy -Dgar.name=$QGS_DIR/src/build/lib/org.globus.cog.qos.Qgs.gar
   where QGS_DIR is the installation directory of this distribution

5. Start the OGSI container by entering the following command from the OGSA directory: ant startContainer

6. Create a persistent instance of the QGS by entering the following command from the <ogsa_dir>/bin: ogsi-create-service \
   http://localhost:8080/ogsa/services/org/globus/cog/qos/QgsService test
   this should be entered as one command.

7. To ensure the service instance has been started, from the ogsa_dir enter the following command: ant gui This command starts the OGSI visual browser. You should see in the browser: 'A QoS Service Factory' and 'A QoS Service Instance' with both in 'ACTIVE' states.

8. If all goes well, and you can see the service instance in the browser as 'ACTIVE', then Congratulations!! – the QGS is deployed and instantiated correctly.

B.3. Bug Reports

To report bugs please use http://www.globus.org/cog/contact/bugs

or e-mail: Rashid Al-Ali at rashid@mcs.anl.gov or rashid@cs.cardiff.ac.uk

Appendix C

DSRT Wrapper API

```
package org.globus.cog.qos.server.dsrtApi;

import org.globus.cog.core.impl.common.CoreProperties;

public class QosDsrtProxy {
    private String cpuPercent;
    private String option;
    private String pid;
    private String dsrtPath = null;
    private String dsrtAPI = "DSRTapi.o";

    public void setDsrtPath() {

        try {
            CoreProperties properties = new CoreProperties("config.txt");
            this.dsrtPath = properties.getCoreProperty("DSRT_INSTALLATION");
        } catch (Exception e) {
            System.out.println(e);
        }
    }


    public void setCpuPercent(int cpuPercent) {
        this.cpuPercent = Integer.toString(cpuPercent);
    }

    public String getCpuPercent() {
        return this.cpuPercent;
    }


    public void setPid(int pid) {
        this.pid = Integer.toString(pid);
    }


    public String getPid() {
        return this.pid;
    }


    public void allocateResource() {
        this.option = "0";
        this.contactDSRT();
    }


    public void releaseResource() {
        this.option = "1";
        this.contactDSRT();
    }


    private void contactDSRT() {
        try {
```

```
            String cmd = dsrtPath + "/" + dsrtAPI + " " + cpuPercent + " " +
              pid + " " + option;

            Runtime rt = Runtime.getRuntime();
            System.out.println("Exec: executing: " + cmd);
            rt.exec(cmd);
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

C.1. DSRT QoS Command Execution – Java Class

```
package org.globus.cog.qos.server.dsrtApi;

import org.globus.cog.core.impl.common.CoreProperties;

import java.io.*;

public class QosExecCommand {
    private String cCodePath = "prog";
    private String executable;
    private String param1;
    private String utilPath;

    public QosExecCommand(String executable, String param1) {
        this.executable = executable;
        this.param1 = param1;

        try {
            CoreProperties properties = new CoreProperties("config.txt");
            this.utilPath = properties.getCoreProperty("DSRT_INSTALLATION")
              + "/util/";
        } catch (Exception e) {
            System.out.println(e);
        }
    }


    public String getCommandArguments() {
        String cmd = null;

        // note: executable + " " + executable ..... this is an Excel
             requirement!
        cmd = utilPath + "pid.txt" + " " + executable + " " + executable + "
             " + param1;
        System.out.println("Service: the generated cmd. Args: " + cmd);

        return cmd;
    }


    public String getCommandExec() {
        String cmd = null;

        cmd = utilPath + cCodePath;
        System.out.println("Service: the generated cmd. Exec: " + cmd);

        return cmd;
    }

    public String getPid() {
```

```
        String pid = null;
        try {
            File file = new File(utilPath + "pid.txt");
            if (file.canRead()) {
                FileInputStream fis = new FileInputStream(file);
                int Cbuffer = -1;
                char buf[] = new char[10];
                int i = 0;
                char C;

                do {
                    Cbuffer = fis.read();
                    C = (char) Cbuffer;
                    if (Cbuffer != -1) {
                        System.out.print(C);
                        buf[i++] = C;
                    }
                } while (Cbuffer != -1);

                StringBuffer strbuff = new StringBuffer();
                strbuff.append(buf);
                String temp = strbuff.toString();
                pid = temp.trim();
                fis.close();
                file.delete();
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
        return pid;
    }
}
```

Appendix D

A Java Class for QoS Negotiation

```
package org.globus.cog.qos.examples;

import org.apache.log4j.Logger;
import org.globus.cog.core.impl.common.*;
import org.globus.cog.qos.handler.QosTaskHandlerImpl;
import org.globus.cog.qos.handler.QoS;
import org.globus.cog.core.interfaces.*;

public class QosRequest2 {
    static Logger logger =
Logger.getLogger(QosRequest2.class.getName());
    private Task task;

    public QosRequest2() {
        prepareTask();
        submitTask();

        String status = (String)
        this.task.getAttribute("agreementToken");

        if (status != null) {
            System.out.println("Your request has SUCCEEDED and the
            agreementID is: " + status);
        } else
            System.out.println("Your request has FAILED!");
    }

    private void prepareTask() {
        String startTime = "11/10/2003 16:21:00";
        String endTime = "11/10/2003 16:35:00";
        String serviceContact =
        "http://localhost:8080/ogsa/services/org/globus/
            cog/qos/server/QosService/qos";
        String allocationStrategy = "resource-domain"; // or can be
            time-domain

        task = new TaskImpl("myTestTask", QoS.QoSNEGOTIATION);
        logger.debug("Task Identity: " +
        task.getIdentity().getValue());

        this.task.setAttribute("startTime", startTime);
        this.task.setAttribute("endTime", endTime);
        this.task.setAttribute("allocationStrategy",
            allocationStrategy);

        if (allocationStrategy.compareTo("resource-domain") == 0) {
            this.task.setAttribute("resourceCapacity", "40");
        }

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

```
        this.task.setServiceContact(service);
    }

    private void submitTask() {
        TaskHandler handler = new QosTaskHandlerImpl();
        try {
            handler.submit(this.task);

        } catch (InvalidSecurityContextException ise) {
            logger.error("Security Exception");
            ise.printStackTrace();
            System.exit(1);
        } catch (TaskSubmissionException tse) {
            logger.error("TaskSubmission Exception");
            tse.printStackTrace();
            System.exit(1);
        } catch (IllegalSpecException ispe) {
            logger.error("Specification Exception");
            ispe.printStackTrace();
            System.exit(1);
        } catch (InvalidServiceContactException isce) {
            logger.error("Service Contact Exception");
            isce.printStackTrace();
            System.exit(1);
        }
    }


    public static void main(String arg[]) {
        new QosRequest2();
    }
}
```

D.1. Submitting a QoS-based Job after QGS Negotiation

```
package org.globus.cog.qos.examples;

import org.apache.log4j.Logger;
import org.globus.cog.core.impl.common.*;
import org.globus.cog.qos.handler.QosTaskHandlerImpl;
import org.globus.cog.qos.handler.QoS;
import org.globus.cog.core.interfaces.*;

public class QosJobSubmission implements StatusListener {
    static Logger logger =
Logger.getLogger(QosJobSubmission.class.getName());
    private Task task;

    public QosJobSubmission() {
        prepareTask();
        submitTask();

        Status jobStatus = this.task.getStatus();
        if (Status.SUBMITTED == jobStatus.getStatus()) {
            System.out.println("Job has been submitted.");
        }

        if (Status.FAILED == jobStatus.getStatus()) {
            System.out.println("Job submission has failed.");
        }
    }

    private void prepareTask() {
        String serviceContact =
        "http://localhost:8080/ogsa/services/org/globus/
            cog/qos/server/QosService/qos";
        this.task = new TaskImpl("myTestTask", QoS.JOB_SUBMISSION);
        logger.debug("Task Identity: " +
        this.task.getIdentity().getValue());

        this.task.setAttribute("agreementToken",
        "localhost.localdomain:1068608841065:120");
        JobSpecification spec = new JobSpecificationImpl();
        spec.setExecutable("/bin/sleep");
        spec.setArguments("30");
        spec.setStdOutput("qosOutput");

        spec.setBatchJob(true);
        this.task.setSpecification(spec);

        ServiceContact service =
                new ServiceContactImpl(serviceContact);
        this.task.setServiceContact(service);

        this.task.addStatusListener(this);
    }

    private void submitTask() {
        TaskHandler handler = new QosTaskHandlerImpl();
```

```
        try {
            handler.submit(this.task);
        } catch (InvalidSecurityContextException ise) {
            logger.error("Security Exception");
            ise.printStackTrace();
            System.exit(1);
        } catch (TaskSubmissionException tse) {
            logger.error("TaskSubmission Exception");
            tse.printStackTrace();
            System.exit(1);
        } catch (IllegalSpecException ispe) {
            logger.error("Specification Exception");
            ispe.printStackTrace();
            System.exit(1);
        } catch (InvalidServiceContactException isce) {
            logger.error("Service Contact Exception");
            isce.printStackTrace();
            System.exit(1);
        }
    }

    public void statusChanged(StatusEvent event) {
        Status status = event.getStatus();
        logger.debug("Status changed to " + status.getStatus());
        if (status.getStatus() == Status.COMPLETED) {
            logger.debug("Output = " + this.task.getStdOutput());
            System.out.println("Job has completed!");
            System.exit(1);
        }
    }

    public static void main(String arg[]) {
        new QosJobSubmission();
    }
}
```

Appendix E

Reservation Data Structure and Methods

```
package org.globus.cog.qos.server.reservation;

import java.util.Date;

/**
 An implementation of a Reservation
 **/
public class QosReservation implements QosReservationInterface {

    Date startTime;
    Date endTime;
    Date submitTime;
    Date lastModified;
    boolean isActive;
    String type;
    String label;
    int capacity;
    String nodeName;
    String id;
    boolean resConfirmed = false;
    String strategy;

    private void modify() {
        Date now = new Date();
        lastModified = now;
    }

    public QosReservation() {
        Date now = new Date();
        lastModified = now;
        submitTime = now;
        isActive = false;
        startTime = now;
        endTime = now;
        label = "undefined";
        capacity = 0;

    }

    public QosReservation(Date from, Date to) {
        Date now = new Date();
        lastModified = now;
        submitTime = now;

        startTime = from;
        endTime = to;

        label = "undefined";
        isActive = false;
        type = "undefined";
        capacity = 0;
```

```
}

public String getLabel() {
    return label;
}

public void setLabel(String l) {
    label = l;
}

/** ----------------------------------------
 Start Time
 ---------------------------------------- **/

/**
 * Get the StartTime when the reservation is set.
 * @return the StartTime of the reservation.
 */
public Date getStartTime() {
    return startTime;
}

/**
 * Set the StartTime for the reservation.
 * @param newStartTime The new StartTime of the reservation.
 */
public void setStartTime(Date newStartTime) {
    modify();
    this.startTime = newStartTime;
}

/** ----------------------------------------
 Type
 ---------------------------------------- **/

/**
 * Get the Type when the reservation is set.
 * @return the Type of the reservation.
 */
public String getType() {
    return type;
}

/**
 * Set the Type for the reservation.
 * @param newType The new Type of the reservation.
 */
public void setType(String newType) {
    modify();
    this.type = newType;
}

/** ----------------------------------------
 Capacity
 ---------------------------------------- **/

/**
 * Get the Capacity when the reservation is set.
 * @return the Capacity of the resource.
 */
public int getCapacity() {
```

```
    return capacity;
}

/**
 * Set the Capacity for the reservation.
 * @param newCapacity The new Capacity of the reservation.
 */
public void setCapacity(int newCapacity) {
    modify();
    this.capacity = newCapacity;
}

/** ----------------------------------------
 EndTime
 ---------------------------------------- **/

/**
 * Get the EndTime when the reservation is set.
 * @return the EndTime of the reservation.
 */
public Date getEndTime() {
    return endTime;
}

/**
 * Set the EndTime for the reservation.
 * @param newEndTime The new EndTime of the reservation.
 */
public void setEndTime(Date newEndTime) {
    modify();
    this.endTime = newEndTime;
}

/** ----------------------------------------
 submit Time
 ---------------------------------------- **/

/**
 * Get the SubmitTime when the reservation is set.
 * @return the SubmitTime of the reservation.
 */
public Date getSubmitTime() {
    return submitTime;
}

/**
 * Set the SubmitTime for the reservation.
 * @param newSubmitTime The new SubmitTime of the reservation.
 */
public void setSubmitTime(Date newSubmitTime) {
    modify();
    this.submitTime = newSubmitTime;
}


/** ----------------------------------------
 last modified
 ---------------------------------------- **/


/**
```

```
 * Get the LastModified when the reservation is set.
 * @return the LastModified of the reservation.
 */
public Date getLastModified() {
    return lastModified;
}

/**
 * Set the LastModified for the reservation.
 * @param newLastModified The new LastModified of the
  reservation.
 */
public void setLastModified(Date newLastModified) {
    this.lastModified = newLastModified;
}


// Node name: is the name of the computer that the reservation
  is made for
public String getID() {
    return id;
}


public void setID(String id) {
    modify();
    this.id = id;
}


public String getStrategy() {
    return this.strategy;
}


public void setStrategy(String strategy) {
    this.strategy = strategy;
}

// a flag to indicate reservation was confirmed or not
public boolean isReservConfirmed() {
    return this.resConfirmed;
}


public void setReservConfirmation(boolean confirmed) {
    this.resConfirmed = confirmed;
}

/** ----------------------------------------
 toXML
 ---------------------------------------- **/


private String field(String name, String value) {
    return name + "=" + value;
}

private String field(String name, Date value) {
    return name + "=" + value.toString();
}
```

```
private String field(String name, int value) {
    return name + "=" + value;
}

private String field(String name, boolean value) {
    return name + "=" + value;
}

/**
 * Returns the Reservation in XML format. Not implemented yet.
 * @return the reservation in XML through a String.
 */
public String toXML(String indent) {

    String out =
            indent + "<reservation" +
            field(indent + "label", label) +
            field(indent + "start", startTime) +
            field(indent + "end", endTime) +
            field(indent + "submitted", submitTime) +
            field(indent + "modified", lastModified) +
            field(indent + "active", isActive) +
            field(indent + "type", "node:" + type + ":" +
              getID()) +
            field(indent + "capacity", capacity) +
            field(indent + "Strategy", strategy) + indent + ">";
    return out;
}

public String toXML() {
    return (toXML(""));
}

/** ----------------------------------------
 activation
 ---------------------------------------- **/

/**
 * Changes the state of the reservation to active
 */
public void activate() {

    modify();
    isActive = true;
}

/**
 * Changes the state of the reservation to deactivate
 */
public void deactivate() {
    modify();
    isActive = false;
}


public int compare(QosReservation r) {
    int result = 0;
    // compares if the other reservation outside of the
        current.
    if (startTime.after(r.startTime) ||
```

```
                endTime.before(r.startTime)) {
            result = 0;
        } else {
            result = -1;
        }
        return result;
    }

}
```

E.1. A Java Class for the Reservation Agent

```
package org.globus.cog.qos.server.reservation;

import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;

public class QosReservationAgent {

    static int id = 111;

    private String label;
    private Hashtable reserveTable;

    public QosReservationAgent(String l) {
        label = l;
        reserveTable = new Hashtable();
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String l) {
        label = l;
    }

    public boolean isAvailable(QosReservation r) throws
      QosReservationException {

        QosReservationValidation resValidation = null;

        resValidation = new
            QosReservationValidation(r.getCapacity(), reserveTable);

        if (!resValidation.validateReservation(r.getStartTime(),
            r.getEndTime())) {
            throw new QosReservationException("Cannot make
            reservation for the given request !!");
        }
        return true;
    }


    public String extend(String label, long durationInMin) throws
            QosReservationException {

        if (durationInMin <= 0) {
```

```
            throw new QosReservationException("Check the supplied
            extension duration !!");
        }
        String reply = null;
        QosReservation r = (QosReservation) reserveTable.get(label);
        long newLongEndTime = r.getEndTime().getTime() +
            durationInMin * (1000 * 60);
        Date newEndTime = new Date(newLongEndTime);
        Date newStartTime = new Date(r.getEndTime().getTime() + 60 *
            1000); //increment by a minute
        QosReservation newR = new QosReservation(newStartTime,
            newEndTime);
        newR.setCapacity(r.getCapacity());
        if (this.isAvailable(newR)) {
            QosReservation extendedR = r;
            extendedR.setLabel(r.getLabel());
            extendedR.setCapacity(r.getCapacity());
            extendedR.setReservConfirmation(true);
            extendedR.setID(r.getID());
            extendedR.setStartTime(r.getStartTime());
            extendedR.setEndTime(newR.getEndTime());
            reserveTable.remove(label);
            reserveTable.put(extendedR.getLabel(), extendedR);
            reply = extendedR.getLabel();
        }
        return reply;
    }


    public String add(QosReservation r) throws
            QosReservationException {

        if (reserveTable.get(r.getLabel()) != null) {
            return null; // this label has been used in another
                        entry
        }
        if (isAvailable(r)) {
            reserveTable.put(r.getLabel(), r);
            return this.createToken(r);
        }
        return null;
    }


    public String delete(String label) throws
      QosReservationException {

        QosReservation r = (QosReservation) reserveTable.get(label);
        Date currentTime = new Date();

        if (r != null) {
            if (!currentTime.before(r.getStartTime())) {
                return ("cannot delete sla");
            } else {
                reserveTable.remove(label);
                return ("successful");
            }
        } else
            return ("failed");
    }
```

```
public String completion(String label) throws
  QosReservationException {

    QosReservation r = (QosReservation) reserveTable.get(label);

    if (r != null) {
        if (this.isTimeToStartTheReservSrvc(label)) { //means
              yes we can report on completion
            reserveTable.remove(label);
            return ("successful");
        }
    }
    return ("failed");
}

public boolean isReservExist(String label) throws
  QosReservationException {

    QosReservation r = (QosReservation) reserveTable.get(label);
    if (r != null) {
        return true;
    }
    return false;
}


public String toXML() throws QosReservationException {

    String result = "";

    result = "reservationAgent ";
    result = result + "name=" + label;
    for (Enumeration e = reserveTable.keys();
        e.hasMoreElements();) {
        QosReservation r = (QosReservation)
        reserveTable.get(e.nextElement());
        result = result + r.toXML("\t");
    }
    return result + "\t>";
}


private String createToken(QosReservation r) {

    String idString = Integer.toString(id++);
    r.setID(idString);
    return (r.getType() + ":" + r.getLabel() + ":" + r.getID());
}


public boolean isTimeToStartTheReservSrvc(String labelin) {

    QosReservation r = (QosReservation)
    reserveTable.get(labelin);
    Date currentTime = new Date();

    if (r != null) {
        if (currentTime.before(r.getEndTime()) &&
                ((currentTime.after(r.getStartTime())) ||
        (currentTime.equals(r.getStartTime())))) {
```

```
                return true;
            }
        }

        return false;
    }


    public String getCapacity(String labelin) {

        QosReservation r = (QosReservation)
        reserveTable.get(labelin);

        return Integer.toString(r.getCapacity());
    }


    public String getStrategy(String labelin) {

        QosReservation r = (QosReservation)
            reserveTable.get(labelin);

        return r.getStrategy();
    }


    public boolean getConfirmationStatus(String label) {

        QosReservation r = (QosReservation) reserveTable.get(label);
        if (r != null) {
            return r.isReservConfirmed();
        } else
            return false;
    }


    public QosReservation getReservation(String label) {

        return (QosReservation) reserveTable.get(label);
    }


    public void deleteReservationEntries() {

        for (Enumeration e = reserveTable.keys();
            e.hasMoreElements();) {
            QosReservation r = (QosReservation)
            reserveTable.get(e.nextElement());
            reserveTable.remove(r.getLabel());
        }
    }
}
```

E.2. A Java Class for Validating Reservation Requests

```
package org.globus.cog.qos.server.reservation;

//import org.globus.cog.qos.impl.QgsImpl;

import java.util.Date;
```

```
import java.util.Hashtable;
import java.util.Enumeration;

public class QosReservationValidation {
    private int reqCapacity, tempCapacity;
    private int durationIterator;
    private Hashtable reservationTable;

    public QosReservationValidation(Hashtable reservationTable) {
        this.reqCapacity =
            Integer.parseInt(QosRequestHandler.MAX_CAPACITY);
        this.reservationTable = reservationTable;
        tempCapacity = 0;
    }

    public QosReservationValidation(int reqCapacity, Hashtable
            reservationTable) {
        this.reqCapacity = reqCapacity;
        this.reservationTable = reservationTable;
        tempCapacity = 0;
    }

    // this method is to check if two given reservation times have
            intersections
    public boolean isWithIn(Date sTref, Date eTref, Date sTService,
            Date eTService) {
        if ((sTService.after(sTref) || (sTService.compareTo(sTref)
            == 0))
                && (sTService.before(eTref) ||
            sTService.compareTo(eTref) == 0)) {
            return true;
        } else if ((eTService.after(sTref) ||
            (eTService.compareTo(sTref) == 0))
                && (eTService.before(eTref) ||
            eTService.compareTo(eTref) == 0)) {
            return true;
        } else if ((sTService.before(sTref) ||
            (sTService.compareTo(sTref) == 0))
                && (eTService.after(eTref) ||
            eTService.compareTo(eTref) == 0)) {
            return true;
        } else
            return false;
    }


    // this method is to convert the period of a reservation into
            minutes
    public int convertDurationToIteration(Date sTime, Date eTime) {
        long duration = ((eTime.getTime() - sTime.getTime()) / (1000
            * 60));
        return ((int) duration);
    }


    // this method is to reset the total capacity of intersected
            services.

    public void resetCapacity() {
        tempCapacity = 0;
    }
```

```
    // this method is to check the total accumulated capacities --
            admission control
    public boolean checkAdmission() {
        int netcapacity =
            Integer.parseInt(QosRequestHandler.MAX_CAPACITY) -
                  this.tempCapacity;
        if (netcapacity >= this.reqCapacity) {
            return true;
        }
        return false;
    }


    public boolean validateReservation(Date sTimeIn, Date eTimeIn) {
        Date sTime = new Date(sTimeIn.getTime());
        Date eTime = new Date(eTimeIn.getTime());

        /* check that the end time is after the start time and
           the start time is later than current time. */
        if ((sTimeIn.after(eTimeIn)) || (sTimeIn.before(new Date()))
                  ||
                (sTimeIn.compareTo(eTimeIn) == 0)) {
            return false;
        }
        this.durationIterator =
            this.convertDurationToIteration(sTime, eTime);
        Date sTService = null, eTService = null;

        int resourceValue = 0;  // a variable to hold resource
            capacity

        for (int i = 0; i < durationIterator; i++) {
            for (Enumeration e = reservationTable.keys();
                  e.hasMoreElements();) {
                QosReservation r = (QosReservation)
                  reservationTable.get(e.nextElement());
                sTService = r.getStartTime();
                eTService = r.getEndTime();
                resourceValue = r.getCapacity(); // SHOULD BE FIXED
                        TO HOLD RESOURCE CAPACITY

                if (this.isWithIn(sTime, eTime, sTService,
                        eTService)) {
                    this.tempCapacity = this.tempCapacity +
                        resourceValue;
                }
            }

            if (!(this.checkAdmission())) {
                return false;
            }
            sTime.setTime(sTime.getTime() + (60 * 1000));
                  //increment by a minute
            this.resetCapacity();     // reset capacity counter
        }
        return true;
    }
}
```

Appendix F

Java Code for Interfacing the QoS Registry Service UDDIe

```
package gqosm.ns.uddie;

import aqos.dataType.*;
import org.uddi4j.*;
import org.uddi4j.client.*;
import org.uddi4j.datatype.*;
import org.uddi4j.datatype.assertion.*;
import org.uddi4j.datatype.binding.*;
import org.uddi4j.datatype.business.*;
import uk.ac.cf.cs.uddie4j.datatype.service.*;
import uk.ac.cf.cs.uddie4j.datatype.service.BusinessServices;
import org.uddi4j.datatype.tmodel.*;
import org.uddi4j.request.*;
import uk.ac.cf.cs.uddie4j.response.eServiceDetail;
import org.uddi4j.response.DispositionReport;
import org.uddi4j.response.BusinessList;
import org.uddi4j.response.AuthToken;
import org.uddi4j.response.BusinessDetail;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.ServiceList;
import org.uddi4j.response.*;
import uk.ac.cf.cs.uddie4j.response.eServiceDetail;
import org.uddi4j.util.*;
import uk.ac.cf.cs.uddie4j.UDDIeElement;
import org.w3c.dom.Element;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.util.Vector;
import java.util.Properties;
import java.io.*;
import uk.ac.cf.cs.uddie4j.client.UDDIeProxy;
import uk.ac.cf.cs.uddie4j.datatype.lease.*;
import gqosm.ns.datatype.*;
import gqosm.ns.util.StatusWindow;

public class UDDIeInterface {

  Service_Request service;
  StatusWindow status;

  public UDDIeInterface(Service_Request service, StatusWindow
      status)
  {
    this.service = service;
    this.status = status;
    status.setCurrentTaskProgressBar(3);
  }

  public UDDIeInterface()
  {
  }
```

```
/**
 * Get Services which match a specific serviceName
 * and service properties
 * @return Vector of relevant Services
 * @throws Exception
 */
public Vector getServices() throws Exception
{
       UDDIeProxy proxy = new UDDIeProxy();
       proxy.setInquiryURL("http://localhost:8080/uddie/inquiry");
       proxy.setPublishURL("http://localhost:8080/uddie/publish");

       //Get Authorization by sending a username and password
       // for the owner of the business
       //AuthToken token = proxy.get_authToken("gqosm", "gqosm");
       AuthToken token = proxy.get_authToken("Rashiduddie",
          "Rashiduddie");

       //Define service name and add them to a vector
       //The maximum allowed names is 5

       Name name = new Name(service.getServiceName());
       Vector names = new Vector();
       names.add(name);
       Vector properties = new Vector();

       status.addSubTask("Creating SOAP message for the requested
          service");
       // Define property and add them to a Vector
       if ( service.getBudget() != null )
       {
         Property property = new Property("budget", "number",
                service.getBudget());
         property.setPropertyFindQualifer(
                PropertyFindQualifiers.LESS_THAN_OR_EQUAL);
         properties.add(property);
       }

       if ( service.getCpu_count() != null)
       {
         Property property2 = new Property("cpu_count", "number" ,
                service.getCpu_count());
         property2.setPropertyFindQualifer(
                PropertyFindQualifiers.GREATER_THAN_OR_EQUAL);
         properties.add(property2);
       }

       if ( service.getReliability() != null)
       {
         Property property3 = new Property("reliability", "number"
                , service.getReliability());
         property3.setPropertyFindQualifer(
                PropertyFindQualifiers.GREATER_THAN_OR_EQUAL);
         properties.add(property3);
       }

       if ( service.getBandwidth() != null)
       {
         Property property4 = new Property("bandwidth", "number" ,
                service.getReliability());
```

```
            property4.setPropertyFindQualifer(
                    PropertyFindQualifiers.GREATER_THAN_OR_EQUAL);
            properties.add(property4);
        }

        // Define a propertyBag and add the properties Vector in
                the Bag
        PropertyBag bag = new PropertyBag();
        bag.setPropertyVector(properties);

        // Define Find Qualifier for property exact match (Logical
                AND)
        FindQualifier findQualifier = new
                FindQualifier("exactPropertyMatch");
        FindQualifier findQualifier2 = new
                FindQualifier("exactNameMatch");
        FindQualifier findQualifier3 = new
                FindQualifier("exactMatch");
        FindQualifiers qualifiers = new FindQualifiers();
        Vector qualifiersVector = new Vector();
        qualifiersVector.add(findQualifier);
        qualifiersVector.add(findQualifier2);
        qualifiers.setFindQualifierVector(qualifiersVector);

        // Send the query
        status.addSubTask("Sending request to UDDIe");
        ServiceList list = proxy.find_eService(null, names, null,
                bag, null, qualifiers , 5);

        ServiceInfos infos = list.getServiceInfos();
        status.addSubTask("receiving reply from UDDIe");
        Vector services = infos.getServiceInfoVector();
        Vector resultServices = new Vector();

        for ( int i = 0; i < services.size() ; i++)
        {
           ServiceInfo service = (ServiceInfo)services.get(i);
           eServiceDetail serviceDetail =
           proxy.get_eServiceDetail(service.getServiceKey());
           Vector serviceVector =
           serviceDetail.getBusinessServiceVector();
           BusinessService returnedService =
           (BusinessService)serviceVector.firstElement();
           resultServices.add(returnedService);
        }
        return resultServices;
}

/**
 *  Return BusinessService Detail based on a
    Request_Specific_Service msg
 */
public  BusinessService
    getSpecificService(Request_Specific_Service sService)
{
  try
  {
    return  getServiceDetail(sService.getServiceKey());

  } catch(Exception exp)
  {
```

```
    System.out.println("Error in UDDIe: " + exp);
  }
  return null;
}

/**
 *  Get the URL address of Service Provider
 *  @parm BusinessKey  the businessKey of the provider
 */
public String getBusinessAddress(String businessKey) throws
    Exception
{
    UDDIeProxy proxy = new UDDIeProxy();
    proxy.setInquiryURL("http://localhost:8080/uddie/inquiry");
    proxy.setPublishURL("http://localhost:8080/uddie/publish");

    BusinessDetail businessDetail =
    proxy.get_businessDetail(businessKey);
    Vector business = businessDetail.getBusinessEntityVector();
    for ( int i = 0; i < business.size(); i++)
    {
      BusinessEntity businessEntity =
                (BusinessEntity)business.get(i);
      Vector urls =
          businessEntity.getDiscoveryURLs()
                .getDiscoveryURLVector();
      for ( int j = 0 ; i < urls.size(); j++)
      {
        DiscoveryURL url = (DiscoveryURL)urls.get(j);
        return url.getText();
      }
    }
  return null;
}

/**
 * Get Service Detail based on the Service Key
 * From UDDIe
 */
public BusinessService getServiceDetail(String key) throws
    Exception
{
    UDDIeProxy proxy = new UDDIeProxy();
    proxy.setInquiryURL("http://localhost:8080/uddie/inquiry");
    proxy.setPublishURL("http://localhost:8080/uddie/publish");

    eServiceDetail serviceDetail = proxy.get_eServiceDetail(key);
    Vector serviceVector =
    serviceDetail.getBusinessServiceVector();
    BusinessService returnedService =
    (BusinessService)serviceVector.firstElement();
    return  returnedService;
}

/**
 * Get the URL of the WSDL Interface for a given Service
 * @param serviceKey
 */
public String getServiceWSDLInterfaceURL(String serviceKey) throws
    Exception
{
```

```
        BusinessService returnedService =
              getServiceDetail(serviceKey);

        //Get the wsdl interface URL from the best selected service
        String url = "";
        Vector bindingTemplateV =
        returnedService.getBindingTemplates()
              .getBindingTemplateVector();
        for ( int i = 0; i < bindingTemplateV.size(); i++)
        {
          BindingTemplate bt =
                    (BindingTemplate)bindingTemplateV.get(i);
          Vector tmodelV = bt.getTModelInstanceDetails()
              .getTModelInstanceInfoVector();
          for ( int j = 0; j < tmodelV.size(); j++)
          {
            TModelInstanceInfo tmodel =
              (TModelInstanceInfo)tmodelV.get(j);
            url = tmodel.getInstanceDetails()
                    .getOverviewDoc().getOverviewURLString();
          }
        }
        return url;
    }
}
```

F.1. Java Code for Selecting the Closest Matched Service

```
package gqosm.ns;

import gqosm.ns.util.StatusWindow;
import aqos.dataType.*;
import gqosm.ns.uddie.UDDIeInterface;
import gqosm.ns.datatype.*;
import aqos.dataType.*;
import uk.ac.cf.cs.uddie4j.datatype.service.*;
import uk.ac.cf.cs.uddie4j.datatype.service.BusinessServices;
import org.uddi4j.datatype.tmodel.*;
import org.uddi4j.request.*;
import uk.ac.cf.cs.uddie4j.response.eServiceDetail;
import org.uddi4j.response.BusinessDetail;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.ServiceList;
import org.uddi4j.response.*;
import uk.ac.cf.cs.uddie4j.response.eServiceDetail;
import org.uddi4j.util.*;
import uk.ac.cf.cs.uddie4j.UDDIeElement;
import org.w3c.dom.Element;
import java.util.Vector;
import org.uddi4j.datatype.binding.*;


import gqosm.ns.datatype.*;

public class ServiceSelector {

  private StatusWindow status;
  private Service_Request service;

  public ServiceSelector(StatusWindow status, Service_Request
      service) {
    this.status = status;
    this.service = service;
  }

  /**
   * Select the best possible service from UDDIe
   * @return Best_Service Message
   * @throws Exception
   */

  public AqosObject getBestService() throws Exception
  {

    //Contact the UDDIe and get the Matched Services to the request
            service
    status.setMainTask("Service Discovery: Contact UDDIe");
      // <-- Demonstration Only
    UDDIeInterface uddie = new UDDIeInterface(service, status);
    Vector servicesVector = uddie.getServices();

    //Get highest weight of the returned services
    double high = 0;
```

```
int selectedServiceIndex = 0;

status.setMainTask("Selecting the best service (highest WA)");
  //
status.setCurrentTaskProgressBar(3);                          //
  <-- Demonstration Only
status.addSubTask("Computing the total importance level");  //

ImportanceLevel imp = new ImportanceLevel(service);

 status.addSubTask("Computing the Weighted Average (WA) for
 every service");
for ( int i = 0 ; i < servicesVector.size() ; i++)
{
     BusinessService returnedService =
     (BusinessService)servicesVector.get(i);
     PropertyBag bag = returnedService.getPropertyBag();
     Vector propertiesFound = bag.getPropertyVector();

     if ( propertiesFound.size() != 0)
     {
       String cpu_count = "";
       String reliability = "";
       String bandwidth = "";
       String budget = "";

       for ( int j = 0 ; j < propertiesFound.size(); j++)
       {
         Property propertyFound =
              (Property)propertiesFound.get(j);

         if (
              propertyFound.getPropertyName()
                    .equals("cpu_count"))
         {
           cpu_count = propertyFound.getPropertyValue();
         }
         else if ( propertyFound.getPropertyName()
                    .equals("bandwidth"))
         {
           bandwidth = propertyFound.getPropertyValue();
         }
         else if ( propertyFound.getPropertyName()
                    .equals("reliability"))
         {
           reliability = propertyFound.getPropertyValue();
         }
         else if ( propertyFound.getPropertyName()
                    .equals("budget"))
         {
          budget = propertyFound.getPropertyValue();
         }
       }


       double serviceW =
       imp.getImportanceLevel(budget,cpu_count, bandwidth,
              reliability);
       if ( high < serviceW )
       {
         high = serviceW;
```

```
                selectedServiceIndex = i;
              }
            }

        }
          status.addSubTask("Selecting the best service based on its WA
                value");

          //Return the best service
          BusinessService returnedService =
          (BusinessService)servicesVector.get(selectedServiceIndex);
          AqosObject bestService = new AqosObject("best_service");

          //Get the wsdl interface URL from the best selected service
          AqosObject wsdlURL = new AqosObject("wsdl_interface");
          String url = uddie.getServiceWSDLInterfaceURL(
                      returnedService.getServiceKey());
          wsdlURL.setValue(url);
          bestService.addElement(wsdlURL);

          //Get the URL address from the best selected service
          AqosObject urlAddress = new AqosObject("url_address");
          String url_address = uddie.getServiceWSDLInterfaceURL(
                      returnedService.getServiceKey());
          urlAddress.setValue(url_address);
          bestService.addElement(urlAddress);

          //Add the ServiceKey to the best_service message
          AqosObject serviceKey = new AqosObject("service_key");
          serviceKey.setValue(returnedService.getServiceKey());
          bestService.addElement(serviceKey);

          PropertyBag bag2 = returnedService.getPropertyBag();
          Vector propertiesFound = bag2.getPropertyVector();

          for ( int j = 0 ; j < propertiesFound.size(); j++)
          {
              Property propertyFound = (Property)propertiesFound.get(j);
              AqosObject node = new
                      AqosObject(propertyFound.getPropertyName());
              node.setValue(propertyFound.getPropertyValue());
              bestService.addElement(node);
          }

        return bestService;
      }

}
```

Appendix G

Bandwidth Broker in DiffServ

A Bandwidth Broker (BB) is important in providing QoS in a DiffServ domain. Traffic entering a DiffServ domain is classified, and conditioned, as a means to enforce DiffServ agreements between domains, at the boundary of the network, and then assigned to different behaviour aggregates, or group of packets with the same code point. The flows entering a domain are classified, based on the DiffServ Code Point (DSCP) value in each packet header. All packets with the same DSCP are treated in the same manner, and belong to the same behaviour aggregate. The core routers forward packets according to the treatment required on the basis of their behaviour aggregate.

The main resource management entity in a DiffServ domain is the BB, which maintains policies and negotiates SLAs with client and neighbouring domains. The interactions of a BB with other components of a DiffServ domain, such as routers and hosts, and the end-to-end communication process in a DiffServ domain are shown in Figure 5.4. This figure shows that when a flow needs to enter the DiffServ domain, or a local user wants to send traffic, the broker is requested to check related SLAs (SLAs associated with flow) and the present traffic condition on the network. The broker decides whether or not to allow the traffic, on the basis of previously-negotiated SLAs, to ensure that new traffic does not violate current SLAs. If there is a new flow, the broker might have to negotiate a new SLA with the neighbouring domain(s) depending on traffic requirements. Once the broker allows the traffic, the edge or leaf router, i.e. the router on the border of the DiffServ domain, needs to be reconfigured. SLA negotiation is a dynamic process that needs to take into account the ever-changing requirements of network traffic. The BB is responsible for admission control, as it has global knowledge of network topology and resource allocation.

Bandwidth Broker Architecture

A BB is a complex entity, comprising four distinguishable parts: *Inter-domain*, *Intra-domain*, *Database* and *User/Application*, as discussed in the following sub-sections and shown in Figure G.1.

Inter-domain: At the inter-domain level, a BB communicates with neighbouring BBs to reserve resources in other domains. A broker needs this communication when the destination of the user's flow – i.e. the resources requested – is in another DiffServ domain. The Internet2 QBone BB Advisory Council proposed the *simple inter-domain BB signalling* (SIBBS) protocol (Teitelbaum *et al.* 1999). The SIBBS protocol follows a request/response model between peer BBs.

Brokers have long-running Transmission Control Protocol (TCP) connections with one another, with TCP providing the basic reliability and flow control for the protocol. Whenever a broker receives a *resource allocation request* (RAR) from another broker, it checks the sender's identity, the route, and the egress router (edge router of the DiffServ domain) for the flow, and the SLA related to the user or flow. On acceptance of a request, if the destination of the flow is not in the broker's domain, it propagates the RAR to the neighbouring broker on the flow path. In this manner, in due course, the RAR contacts the BB with the destination host in its domain. A *resource allocation answer* (RAA), the response to the RAR, is sent back from the destination broker to the source broker.

Intra-domain: At the intra-domain level, a BB needs to communicate with edge routers as well as core routers, to transmit policy decisions, with the routers configured to provide network QoS. There are many suitable intra-domain protocols, such as COPS (RAP, 2000), SNMP and Telnet; however, intra-domain protocols used in the DiffServ domain are only significant to the local network provider.

COPS is used to send policy decisions from the *policy decision point* (PDP) to the PEP at which IP traffic is handled, and policy-based admission control for data flows is implemented. The PDP has a complete view of the network and configures its PEPs according to network policies. A BB normally has the functionality of a PDP,

with all the edge routers configured as PEPs. COPS is a client-server protocol in which the server – a PDP – has a TCP connection with all its clients – PEPs (Durham *et al.* 2000). A PEP maintains a *policy information base* (PIB), as described by Chan *et al.* (2003).

For supporting policy specifications, a new *client-type COPS for PRovisioning* (COPS-PR) is introduced in Chan *et al.* (2001). A COPS-PR supports real-time event-driven communication. A PEP has only one connection to a PDP in the area of policy control, which supports atomic transactions of data and only exchanges differential updates.

On initialisation, a PEP establishes a connection with a PDP and sends all device-relevant information. The PDP replies with all provisioned policies relevant to the device. If there are any changes in policies at the PDP it sends an update message. Alternatively, if there is a change at the PEP, it sends the change to the PDP which can in turn reply with new relevant policy provisioning elements.

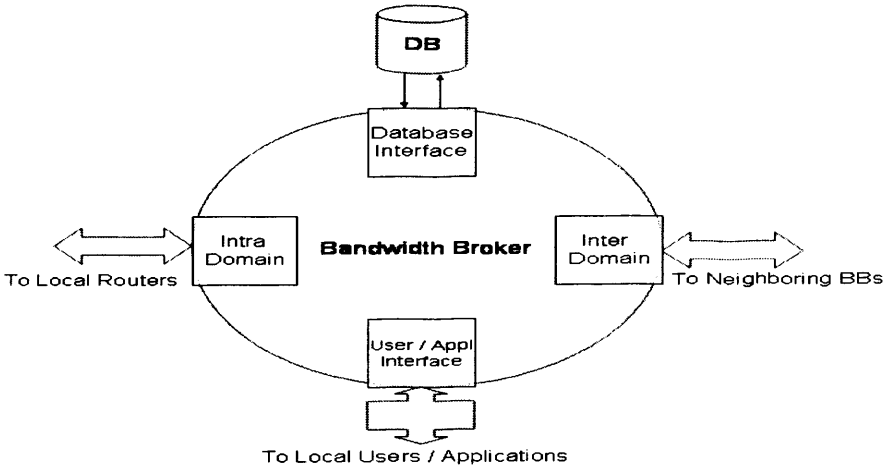

Figure G.1: Bandwidth Broker Concept

Database: A BB has a database interface to gather information for decision-making. To provide QoS, the BB must have a comprehensive picture of the complete network, and needs information on SLAs, network state and current resource allocation status (Teitelbaum *et al.* 1999). Routers can be configured to provide monitoring data, to enhance the security of the network and to improve resource

usage. Routers' configuration data and information about BB's own components is also maintained for the purpose of fault tolerance. Many database management systems are available that can meet a BB's database requirements, such as MySQL and Oracle.

User/Application: There is a need for a protocol and interface for a network operator and/or an application to interact with the BB. A network operator may use this interface to monitor or update performance-related features of a BB, while an application requires the protocol and interface to query a BB.