THE UNIVERSITY OF CHICAGO

END-TO-END QUALITY OF SERVICE FOR HIGH-END APPLICATIONS

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

ALAIN ROY

CHICAGO, ILLINOIS

AUGUST 2001

Dedicated to my wonderful wife, Annalisa.

ABSTRACT

Many computing applications demonstrate increasingly voracious appetites, consuming ever more resources. When high performance applications are required to share networks, computers, and disks with other applications, their performance suffers. When the resources cannot be increased, applications must either adapt or the resources must guarantee better performance to some applications. This latter solution is known as quality of service, or QoS.

This dissertation presents the design and implementation of an architecture to provide end-to-end QoS for high-end applications—those applications consuming a variety of resources and expecting high-performance. End-to-end QoS involves not only traditional network QoS, but other types of QoS, such as CPU and disk, to ensure that the application receives the performance it needs. The major contribution of this work is the design of an innovative and extensible architecture which provides uniform access to different types of QoS.

ACKNOWLEDGEMENTS

I would like to thank my advisor Ian Foster, who was amazingly patient, helpful, and supportive and my other Ph.D. committee members, Mike O'Donnell and Klara Nahrstedt who were dedicated and insightful. This dissertation would be a mere shadow of what it is without Volker Sander's amazing technical assistance. Linda Winkler was always helpful when I needed help with GARNET, our QoS testbed. Steve Tuecke, Karl Czajkowski, and Carl Kesselman assisted greatly in the early design of GARA. Bob Olson was always ready to help demonstrate GARA. Without Nick Karonis and Brian Toonen, my extensions to MPI to support QoS would never have been accomplished. Jenny Schopf provided support and office space which not only saved me many hours of commuting, but offered a fun environment. Bob Brown helped me to see the magic in life. My parents, Michel and Judy Roy, kept me going when my resolve faltered. My wife Annalisa deserves more thanks than I know how to express. And to all of my friends, relatives, co-workers at Argonne National Laboratory, and support staff at the University of Chicago, Argonne, and Northwestern—thank you for all of your help through the years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Many computing applications demonstrate increasingly voracious appetites, consuming ever more resources. While USENET consumed large amounts of bandwidth in the 1980's, multimedia web downloads consume far larger amounts of bandwidth today. Similarly, scientific programs used to measure their speed in megaflops, but now strive for teraflops and process terabytes instead of gigabytes.

Just as data seems to expand to fill any size hard drive, today's most demanding applications strain the capacities of the networks, computers, and storage devices they use. When these applications must share their resources with other applications, they may be unable to perform to the satisfaction of their users. The problem here is twofold: the resources are limited and the amount of a resource available to a particular application fluctuates depending on conditions beyond its control.

If an application does not have enough resources available to meet its performance needs, the only solutions are either to increase the capacity of the resources or to decrease the need for the resources. However, sometimes resources have sufficient capacity for one application, but the actual capacity available to that application fluctuates because the resources are being shared with other applications. The most common example of this is a network, which is almost always shared between multiple applications. If we have such a shared resource and we cannot reliably get a constant and sufficient service from it, there are two general strategies we can use. First, an application can adapt to the amount that is available. For example, a video streaming application may decrease the resolution of the video it sends when less bandwidth is available. Second, the resource may provide a guarantee that it will provide a certain quality, such as a specific bandwidth, to the application. When a resource is able to offer such a guarantee, it is said to offer *quality of service*, or *QoS*.

Applications have varying resource requirements. Applications that are relatively undemanding are often capable of easily adapting or may need only a single type of QoS, such as network QoS. On the other hand, *high-end* applications are very demanding and run in complex environments. They may require combinations of several types of QoS including network, CPU, and storage. Managing multiple resources with QoS can be difficult for applications because each type of QoS is typically controlled by a completely different system with different interfaces, capabilities, and behavior. Yet this management is essential, because without combining different types of QoS, applications may fail to operate well enough to meet users' expectations. Therefore, a major goal of this work is to make it easy for applications to work with different types of QoS:

It is our goal to provide useful, convenient QoS to high-end applications.

The rest of this dissertation describes this goal in more depth, and our design and implementation of a system to accomplish that goal.

The main contribution of this dissertation is a modular and extensible QoS system architecture (GARA) that integrates different QoS mechanisms. This architecture is not merely a simple blending of mechanisms, but an interesting contribution in its own right.

This dissertation also describes two other contributions. First, we have provided significant examples of how to simplify access to QoS: We have extended an implementation of the widely-used Message Passing Interface (MPI) to allow programmers to easily request network QoS, and we have demonstrated methods of combining multiple reservations. Second, we have added to the understanding of mechanisms that can be used to provide QoS, particularly for network QoS.

The combination of these three contributions demonstrates that it is possible to provide a unified QoS system that is convenient for programmers to use and provides a useful capability to high-end applications.

There are many difficulties along the path to this goal. In the next chapter, we explore what this goal means in depth: what QoS is, what high-end applications are, and what makes QoS difficult for high-end applications.

CHAPTER 2
BACKGROUND

Recall that we stated in the previous chapter that our goal is to provide “useful, convenient QoS to high-end applications,” and that QoS is a guarantee from a resource to provide a particular quality, such as bandwidth, to an application. We now look at this goal in more depth by discussing resources; reservations, which encapsulate QoS guarantees; types of QoS; high-end applications; and why high-end applications make QoS more complicated.

2.1 Resources

We begin by defining what a resource is. Pinning down an exact meaning for a resource is challenging, but the definition provided by [54] will serve our needs well:

> ...we will refer to the objects granted [by the operating system] as resources. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a data base).

By this definition, all of the following are resources:

- an entire computer
- a single CPU
- a network
- a network interface
- a storage system
- a hardware graphics pipeline

- a database record
- a software license
- a portion of memory

We consider all of these resources to be candidates for QoS reservations.

2.2 Reservations

When an application wants a resource to provide QoS, it usually has to not only ask for the quality that is needed, but must also specify at what time the QoS is needed. A *reservation* is a guarantee for a certain level of quality from a resource for a particular time period.

In much of the earlier QoS research, reservations did not match the non-computer scientist's view of a reservation. When I make a reservation at a hotel, I normally make it before I arrive at the hotel, and I inform the hotel how long I will stay. While I can make a reservation when I show up at the hotel, I run a greater risk of being turned down. However, many network reservation systems [18, 4] provide only *immediate reservations*. That is, an application could only make a reservation that would begin at the same time the request was made, and the reservation lasted indefinitely, not for a specific duration.

In contrast to immediate reservations, a QoS system could also provide advance reservations, in which the reservation is requested before it is needed, as in our hotel example. Advance reservations are more demanding on the underlying system since they require a larger number of reservations to be tracked. However, they provide important functionality for people who wish to plan their resource usage. Most people would be unhappy to show up for an important software demonstration, only to find that they are unable to use the computers or networks involved in the demonstration. They would rather make a reservation a week in advance for the computers and networks they will need, and if the reservation fails they have time to find alternative resources.

Although immediate reservations were described as having indefinite duration, we can also have immediate reservations with definite duration. In general, we can categorize the time-related aspects of reservations according to (a) when the reservation is made (immediate or in advance), and (b) how long it is made for (a definite time period, or an indefinite time period). This categorization yields four distinct combinations of reservation. In most QoS architectures, reservations are immediate and indefinite.

In practice, it is difficult to have a system that provides both advance and indefinite reservations because many advance reservations may be rejected even though the resources will not be in use during the time requested for the advance reservations. For example, assume that a network can provide 10 Mb/s of reserved bandwidth at any given instant. If there are currently five 2 Mb/s indefinite reservations made for video conferences, we cannot make any advance reservations, even though the current video conferences are not likely to last until the end of time. While there are ways to combine advance and indefinite reservations [16], this remains a problem, although not necessarily a technical problem. For example, some implementations could allow a system administrator to make a policy decision that indefinite actually means “twelve hours or less”, or that indefinite reservations can be preempted by definite reservations.

Note that our GARA architecture, described in Chapters 4 and 5, uses only definite reservations, but allows either immediate or advance reservations.

2.3 Quality of service

Now that we have defined reservations and resources, we can discuss the different types of quality of service that we can provide.

If any of the resources we defined in Section 2.1 are shared between different applications, it is possible to give some applications guaranteed portions of the capacity of the resources. This is what we mean by quality of service. Some specific examples follow.

2.3.1 Network quality of service

Network QoS usually specifies up to four characteristics of data transmission: bandwidth, loss rate, delay, and jitter. *Bandwidth* is the total amount of data sent on the network per time unit (usually, but not necessarily, seconds). Note that bandwidth usually refers to network bandwidth, not the throughput observed by the application. The maximum application throughput is lower than the network bandwidth due to network and protocol overheads, such as packet headers. *Loss rate* is the maximum number of packets that can be lost per time unit. *Delay* is the maximum amount of time that a packet will take to travel from the sender to the receiver. *Jitter* is the maximum variance in delay between successive packets. Limiting the jitter is useful for real-time multimedia applications that wish to present a user with a steady frame rate.

Some people have envisioned other types of network QoS such as security, which may allow flows with different levels of encryption or resistance to tampering, but this is not usually considered, and we will not consider it here.

Note that most network QoS architectures provide half-duplex reservations. That is, if an application requires bi-directional reservations for QoS, two reservations must be made, one for each direction.

2.3.2 CPU quality of service

CPU QoS can be divided into two different categories: shared and exclusive. On a computer where a CPU is shared with different user-level applications, a user might specify a QoS reservation as a percentage of the CPU time over a time interval. On a larger multiprocessor machine there may be only one user-level application per CPU, so a QoS specification may request exclusive access to one or more of the CPUs on a computer.

2.3.3 Storage quality of service

There are two common types of QoS for storage devices: bandwidth and space. *Bandwidth* is the total rate of data sent between the storage device and the application, while *space* is the total amount of data the application can store for any files that it uses.

2.4 High-end applications

It is easiest to understand what we consider to be a high-end application by providing examples. Below we describe three representative examples of the high-end network applications that are encountered, for example, in advanced scientific and engineering computing [21].

2.4.1 Distance visualization of large data sets

Scientific instruments and supercomputer simulations generate large amounts of data: tens of terabytes today, petabytes within a few years. Remote interactive exploration of such data sets requires that the conventional visualization pipeline be decomposed across multiple resources. A realistic configuration might involve moving data at hundreds or thousands of Mb/s to a data analysis and rendering engine which then generates and streams real-time MPEG-2 (or perhaps HDTV, in the future) video to one or more remote client, with control information flowing in the other direction. QoS parameters of particular interest for this class of application include bandwidth, latency and jitter; resources involved in delivering this QoS include storage, network, CPU, and visualization engines.

2.4.2 Large data transfers

In other settings, large data sets are not visualized remotely but instead are transferred in part or in whole to remote sites for storage and/or analysis. The need to coordinate the use of other resources with the completion of these multi-gigabyte or terabyte transfers leads to a need for QoS guarantees of the form “data delivered by deadline” rather than instantaneous bandwidth. Notice that achieving this goal requires the scheduling of storage systems and CPUs as well as networks so as to achieve often extremely high transfer rates.

2.4.3 High-end collaborative environments

High-end collaborative work environments involve immersive virtual reality systems, high-resolution displays, connections among many sites, and multiple interaction modalities including audio, video, floor control, tracking, and data exchange. For example, the NCSA Alliance “Access Grid” currently connects tens of sites via multiple audio, video, and control streams, with the audio streams especially vulnerable to loss. Such applications require QoS mechanisms that allow the distinct characteristics of these different flows to be represented and managed [12].

2.5 Providing QoS to high-end applications

High-end applications such as the ones just described have various QoS requirements, which we now examine.

2.5.1 Heterogeneous network flows

The applications of interest frequently incorporate multiple network flows with widely varying characteristics, in terms of bandwidth, latency, jitter, reliability, and other requirements. For example, a collaborative environment might have low bandwidth control flows but require them to be very low delay, while simultaneously having a high bandwidth video stream that can tolerate high delay.

2.5.2 High bandwidth flows

Some applications involve high bandwidth flows that may require a large percentage of the available bandwidth on a high-speed link. For example, in recent work, Rebecca Nitzan and Brian Tierney of Lawrence Berkeley National Laboratory (LBNL) demonstrated transfer rates of up to 450 Mb/s over a wide area OC12 network [45], and the Globus Project has demonstrated an average wide-area transfer rate of 512 Mb/s with peaks of up to 1.55 Gb/s [1]. The need for high bandwidth has significant implications for both mechanisms and policy. QoS mechanisms are required that can support such flows while allowing coexistence with other flows having different characteristics. At the policy level, we believe that approaches are required that allow for the coordinated management of resources in multiple domains, so that virtual organizations (e.g., a scientific collaboration) can express policies that coordinate the allocation of the resources available to them in different domains.

2.5.3 Need for end-to-end QoS

Satisfying application-level QoS requirements often requires the coordinated management of resources other than networks: for example, a high-speed data transfer can require the scheduling of storage system, network, and CPU resources.

2.5.4 Need for application-level control

Good end-to-end performance requires applications to discover resource availability, to monitor achieved service, and to modify QoS requests and application behavior dynamically.

2.5.5 Need for advance reservation

Specialized resources required by high-end applications such as high-bandwidth virtual channels, scientific instruments and supercomputers are scarce and in high demand; in the absence of advance reservation mechanisms, coordination of the necessary resources is difficult. Advance reservation mechanisms are needed to ensure that resources and services may be properly scheduled.

2.6 The solution

Earlier we stated that our goal is to provide useful, convenient quality of service (QoS) to high-end applications. We can now restate this goal as:

It is our goal to provide high-end applications with the ability to make and control multiple coordinated advance reservations for end-to-end QoS on heterogeneous resources including networks, CPUs, and storage devices.

Broadly speaking, this goal requires that we solve three different and equally important subproblems: the development of low-level QoS mechanisms, the design of a QoS system architecture to conveniently manage these QoS mechanisms, and methods of simplifying access to QoS within high-end applications. This dissertation describes our contributions to all three of these aspects, but the emphasis is on the design and implementation of our QoS architecture, GARA.

- *QoS Architecture* We developed an extensible, modular and novel architecture (GARA) to enable advanced usage of different QoS mechanisms. GARA, described in Chapters 4 and 5, provides advance and immediate reservations of definite duration, feedback to applications, and uniform access to different types of QoS mechanisms.

- *QoS Mechanisms* In Chapters 6 and 7, we describe our implementation of mechanisms to provide network QoS as well as our integration with QoS mechanisms developed by other researchers.

- *High-Level Uses* An important aspect of a supposedly general-purpose architecture is the ease with which it can be integrated naturally into a variety of higher level programming abstractions. We developed such an integration, described in Chapter 7, to assist scientific programmers using the Message Passing Interface (MPI) library to make it easy to make and use network reservations. This work shows that QoS can be integrated into the MPI model with minimal changes to MPI, while still providing convenient access to programmers. Moreover, it shows that our architecture can easily be used to support high-level programming abstractions. In Chapter 7 we also describe other high-level uses of GARA, particularly mechanisms for making multiple reservations, an essential feature for high-end applications.

The rest of this dissertation describes these contributions in more detail.

CHAPTER 3
RELATED WORK

In this chapter, we discuss various QoS system architectures which are comparable to our system GARA (described in Chapters 4 and 5). In addition to complete system architectures, we could discuss the wide variety of research on various QoS mechanisms. However, because the main contribution of this dissertation is the unifying architecture, we have chosen to focus our discussion of related work on complete QoS architectures.

3.1 OMEGA

OMEGA, [41, 40] is a QoS architecture that aims to provide end-to-end QoS guarantees in networked multimedia systems. The creators of OMEGA realized that end-to-end QoS requires not just network QoS, but also CPU and memory QoS, to ensure that the network QoS is effective.

To simplify QoS requests, users do not interact with the underlying reservation systems, but instead interact with the QoS broker, shown in Figure 3.1. The QoS broker serves two important functions:

- The QoS broker translates high-level QoS requests such as “I want to send video that is 120x60 at 20 frames per second” into the appropriate underlying QoS needs, such as bandwidth and loss rates.
- The QoS broker communicates with the various underlying reservation systems on behalf of the user, sparing the user the complexities of interacting with the various reservation systems.

There are a few of important differences between the OMEGA system and our QoS framework, GARA.

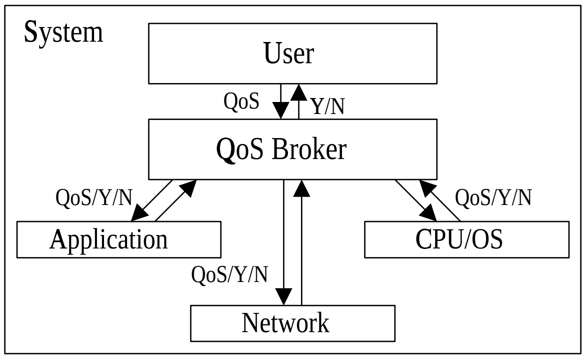

Figure 3.1: The OMEGA QoS Broker. This figure is copied from [41].

- The OMEGA system integrates different types of QoS only in support of network QoS. GARA is more flexible, helping applications to use whatever type of QoS they desire, independently or in combination with other types of QoS. To do this, GARA supports an expandable architecture that adapts easily to different underlying QoS systems.

- OMEGA is focused on supporting multimedia applications, and therefore can incorporate a QoS broker that translates application QoS needs to lower-level QoS needs. Although it is clear how to make this translation for many multimedia systems, as the QoS broker does, it is not clear how to do this in general for high-end scientific applications, so GARA does not provide this sort of translation service. Note, however, that GARA's modular architecture can easily support any translation service that is developed.

- OMEGA, unlike GARA, requires users to use a special library for network communication instead of standard interfaces. This is beneficial because OMEGA can then ensure that the communication processing uses the CPU reservation. However, it places an extra burden on programmers who wish to modify their programs as little as possible in order to receive QoS, and who want their programs to run without modification when QoS mechanisms are not available.

3.2 QoS-A

The QoS Architecture (QoS-A) [6] is an architecture developed by Andrew Campbell to provide end-to-end QoS. Because network guarantees alone are often insufficient, QoS-A is a complete architecture (although only partially implemented) that demonstrates how to realize end-to-end QoS guarantees using a combination of different types of QoS. QoS-A is shown in Figure 3.2. Note that the distributed systems platform and orchestration layers were not implemented, but the lower layers were.

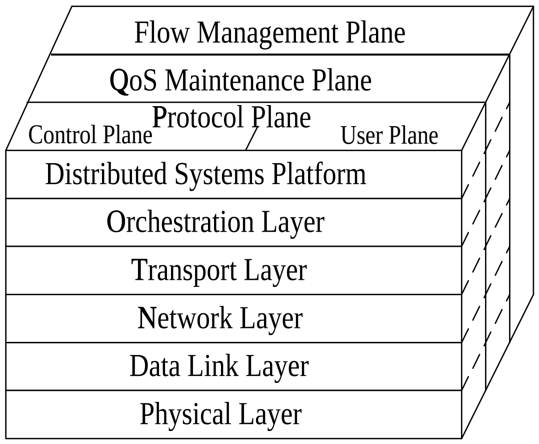

Figure 3.2: QoS-A. This figure is copied from [6].

QoS-A provides both hard (tightly bounded) and soft (more variable) end-to-end guarantees, in addition to best-effort delivery. Hard guarantees are for applications that are not capable of adaptation and have strict performance needs, while soft guarantees are usable by applications that adapt to small variations in service.

Applications that want to have guarantees are required to use a specialized communication protocol called METS to transmit data and a signaling protocol called METSig to make reservations. The guarantees are implemented using ATM, but QoS-A also provides thread scheduling, flow shaping on the end systems, buffer management, and jitter correction to help ensure that the guarantees are met.

QoS-A is a detailed architecture, although only parts of it were fully implemented. Like OMEGA, it requires programmers to use special interfaces for network communication, and is able to provide good guarantees. QoS-A is also similar to OMEGA in that it focuses on networked multimedia applications. Although it can be applied to non-multimedia programs, it is not particularly appropriate for programs that do not require network QoS but do wish for other types of QoS, such as CPU and disk QoS.

3.3 2K and $2K^Q$

$2K^Q$ [42] is a part of the 2K system. It is a unified QoS framework, implemented as a CORBA service. It provides ways to translate end-user QoS parameters, select resources, and make reservations for end-to-end QoS. $2K^Q$ accepts as input a functional graph of the application, rules to translate QoS specifications such as "High Quality" (as provided by an unknowledgeable program user) to specific parameters such as "30 frames per second", and descriptions of the resources that can be used. $2K^Q$ then compiles these three user inputs into specific QoS parameters for specific resources. With such a system, one can presumably translate QoS needs for any particular application.

$2K^Q$ provides a good mechanism for co-reservation (see Chapter 7), which the developers call multi-resource reservation [58]. Users provide graphs describing the connections between components of their programs. $2K^Q$ uses an algorithm to select resources that will fulfill the users request while having the least impact on the system, therefore allowing the maximum number of requests to be fulfilled. It does this by avoiding resources that cause the greatest bottlenecks. This requires, of course, that there are multiple resources available to fulfill users' requests.

3.4 QuO

QuO (Quality of Service for CORBA Objects) [59] takes a different approach to QoS than the previously described systems. The developers of QuO came from a background of building distributed object-oriented applications. They found that applications developed for use on local systems and local area networks (LANs) did not work well in wide area networks (WANs). It was not merely that the applications needed QoS guarantees, but that they could not easily adapt to changing conditions in a WAN because there was a single implementation for an object. If there were multiple implementations, an appropriate one could be chosen for the current conditions.

QuO is a CORBA-based QoS system that provides three important features:

- QuO provides *connections* between clients and the objects that they use. Objects can specify *regions* of quality and provide different implementations when the quality is different. For example, a method might use a different algorithm in a high bandwidth environment than a low bandwidth environment. This allows applications to easily adapt to various conditions at runtime, because they can switch from region to region.

- QuO makes the design decisions of an object explicit so that good adaptation can occur. For example, the structure description language describes the resources that an object requires.

- Users specify QoS needs at a high-level. For example, instead of requesting 5 Mb/s, a programmer will request 1000 remote object invocations per second. Applications do not need to worry about the low-level QoS implementations QuO uses, such as RSVP or differentiated services for network QoS.

QuO is an excellent example of providing useful QoS to high-level programmers that need QoS but do not wish to be burdened with all of the low-level details, and desire to develop programs that can run flexibly in different environments. However, QuO does require that applications use object-oriented programming and CORBA.

CHAPTER 4
GARA, A FRAMEWORK FOR USING QUALITY OF SERVICE

The major contribution of the work presented in this dissertation is the design of the General-purpose Architecture for Reservation and Allocation (GARA) system. GARA provides programmers and users with convenient access to end-to-end quality of service (QoS) for computer applications. It provides uniform mechanisms for making QoS reservations for different types of resources, including computers, networks, and disks. (See Chapter 2 for more information on QoS and reservations.) These uniform mechanisms are integrated into a modular structure that permits the development of a range of high-level services.

4.1 High-level overview

GARA is a straightforward system, as illustrated in Figure 4.1. A GARA-enabled program makes a request for a QoS reservation to the GARA *Arbitrator*. This request is specified in a uniform way: requests are not substantially different for different types of resources and it is easy to make many such requests. The GARA Arbitrator communicates with a *resource manager*. Resource managers may be part of the GARA system, or may be provided by other systems: the GARA Arbitrator ensures seamless communication in either case. The resource manager decides if a QoS reservation can be granted or not. Assuming it can be granted, the GARA Arbitrator returns an opaque string, called a reservation handle, to the requesting application. This reservation handle can be used to manipulate the reservation by modifying, canceling, or querying it.

Note that reservations do not have to be made by the program that will use the reservations. This is important since reservations can be made well in advance of the time that they will be used. However, it is also convenient to be able to make reservations for programs for which you cannot change the source code.

In addition to this basic interaction, GARA provides asynchronous feedback to programs. That is, when there are changes to a reservation, programs can be immediately informed of the change. Feedback ranges from notification that a reservation has begun or ended to notification that a reservation is apparently too small for the application's needs.

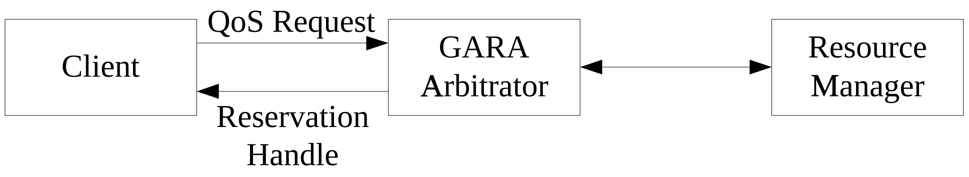

Figure 4.1: A program makes requests to the GARA Arbitrator which mediates access to the resource manager. If a reservation can be made, the GARA arbitrator returns a *reservation handle* describing the reservation.

4.2 Main features of GARA

There are four important features to the GARA architecture:

First, *GARA provides a single, unified interface to advance and immediate definite reservations for a diverse set of underlying resources*. The GARA interface is the same whether the programmer is working with network reservations, CPU reservations, or another other type of reservation. This is true even though the resource managers that implement these reservations work differently and have different interfaces and requirements. We have verified GARA's generality by integrating it with a wide range of resource manager created by ourselves and others. Because of its uniform interface, GARA simplifies the task of working with diverse sets of resource reservations. Note that other comparable systems either do not provide advance reservations, or do not provide a unified interface, or both.

Second, because GARA provides a unified interface to diverse types of QoS, *it is easy to build higher-level services on top of GARA*. An example of such a higher-level service in Section 7.3, where we describe a service that provides convenient mechanisms for creating and manipulating multiple reservations. Another example is in Section 7.4, where we describe how we provide QoS to scientific applications that use the MPI communication library.

Third, *the layered GARA architecture allows for easy extensions as new resource types become available.* For example, a graphic application that makes CPU and network reservations can easily add reservations for graphic pipelines if that ability is added to the lower layers of GARA. It is easy to add to the lower layers, and it does not require deep understanding of the higher layers. (In fact, a collaborator has added the ability to make reservations for graphic pipelines on SGI computers [13].)

Fourth, GARA uses a security infrastructure so that *all reservation requests are securely authenticated and authorized*. Security is an important aspect for a system that allows reservations, yet many QoS systems do not provide security.

4.3 GARA architecture

A view of the abstract GARA architecture can be seen in Figure 4.2.

As shown in the figure, GARA consists of three layers:

- *High-level services layer*
- *Arbitration layer*
- *Resource management layer*

We will describe each of these layers in turn.

4.3.1 High-level services layer

The high-level services layer encompasses a wide variety of services. Because there is a standard interface to the arbitration layer (described in Section 4.3.2), it is easy to build various services that use GARA. This is because the high-level services do not need to concern themselves with the details of the various underlying reservation mechanisms, but can instead concentrate on the strategies that they can provide to users.

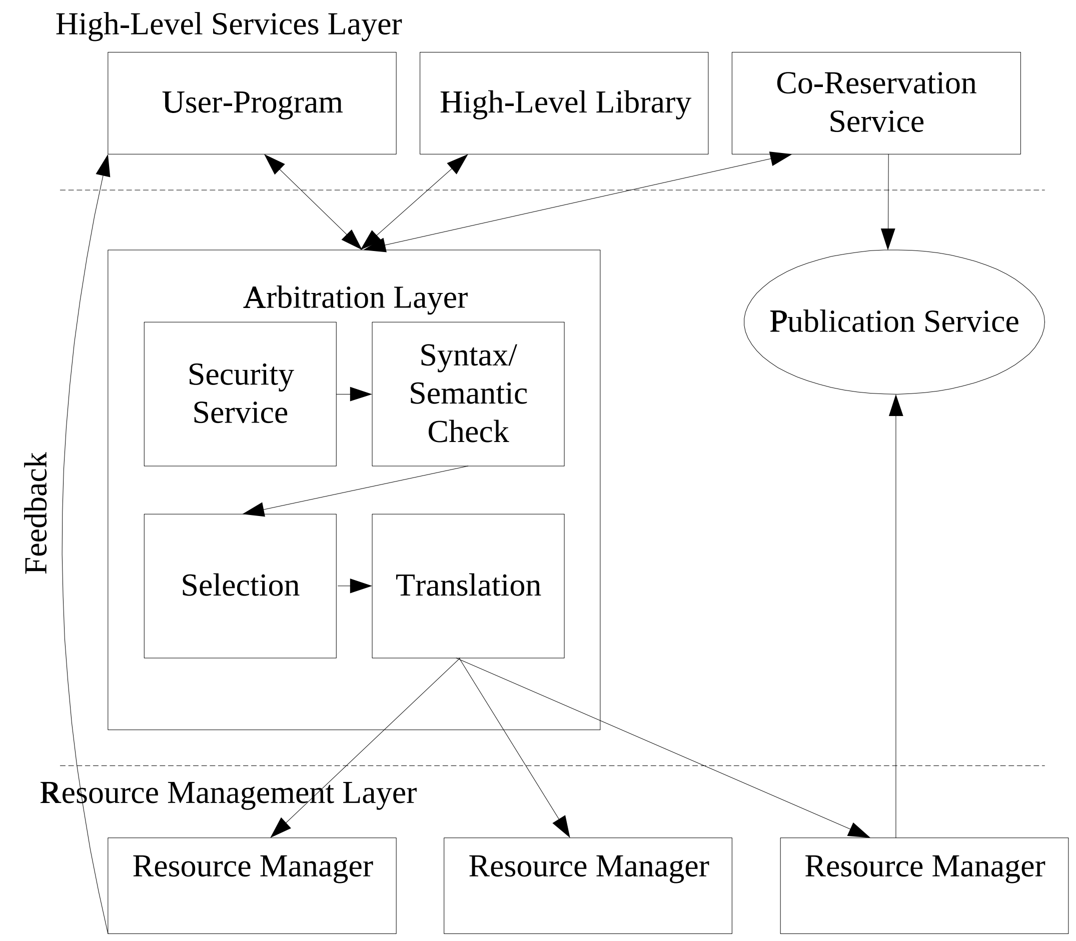

Figure 4.2: An abstract picture of the GARA system architecture. Note that all resource managers can provide feedback and communicate with the publication service, but to keep the diagram simple, not all arrows were drawn.

A variety of high-level services are possible. For example, in Section 7.3 we describe co-reservation services that make a combination of reservations on behalf of users, while the normal GARA interaction via the arbitration layer only allows a user to make a single reservation at a time. Although working with a small number of reservations is a reasonable task for a user, when the number of reservations grows large they become difficult to manage. This is particularly true if resources are scarce and it takes a significant amount time to find appropriate resources that can grant the reservations that the user requires. It is therefore appropriate to create reusable co-reservation agents to encapsulate strategies for discovering resources and making reservations for users.

Another example of a high-level service is integrating QoS into a high-level library. We describe such a service in Section 7.4, which extends an implementation of the Message Passing Interface (MPI), a communication library commonly used by scientific programmers, to allow programmers to easily access reservations for network quality of service. The programmers do not need to understand the details of using GARA, but instead work within the comfortable MPI model that they are used to.

4.3.2 Arbitration layer

In Section 4.1, we said that GARA provides a unified interface to a diverse set of underlying resources. The Arbitrator, or arbitration layer, is the layer that provides this unified interface and interacts with these diverse resources.

Clients communicate with the arbitration layer using the GARA Application Programmer's Interface (API), which is described in detail in Appendix A, but below we provide a brief description of how it works. When clients make requests, the arbitration layer acts as an intermediary to the underlying resource managers (described in Section 4.3.3) that track and manage reservations and resources. The arbitration layer ensures that the client's requests are correct and translates these requests so that they can be understood by the appropriate resource manager. When errors occur they are communicated directly back to the user.

Interacting with the arbitration layer begins when the client makes requests to the arbitration layer. Therefore, we begin by describing the GARA API, which the client uses to make these requests.

The GARA API

Clients use the GARA API to communicate with the arbitration layer. That is, all parameters to make and modify reservations are provided to the arbitration layer through the API, and all errors that may occur in the arbitration layer or the resource management layer are communicated back the client through the API. In this section, we briefly describe the GARA API at an abstract level. The details of the API can be found in Appendix A.

The GARA API has three significant features:

- Clients describe the reservations they wish to make using a list of attribute-value pairs in a text format called the resource specification language, or RSL [11]. The representation that is used is an extensible and easy-to-use format: it is easy to add new attributes, and is easy for the arbitration layer to parse and translate the attributes into appropriate data structures for the underlying resource managers. Moreover, many attributes, such as the time the reservation starts, are applicable to all types of reservations, simplifying the task of a programmer that needs to deal with multiple types of reservations. An example reservation description for a network reservation might look like this:

  ```
  &(reservation-type=network)
    (start-time=953158862)
    (duration=3600)
    (endpoint-a=140.221.48.146)
    (endpoint-b=140.221.48.106)
    (bandwidth=150)
  ```

  Note that clients always specify both a start time and a duration for their reservations. The start-time can be *now* for an immediate reservation, or can be a time in the future, specified as seconds since January 1, 1970. The duration, also specified in seconds, is always required. That is, GARA only uses definite reservations. The endpoints are the computers that will be using the reservation, and the bandwidth is in kilobits per second.

  Also note that the GARA API only allows a single reservation for a single resource to be specified per RSL description. For example, a client can make a reservation for CPU time or a reservation for network bandwidth, but if both are needed the client needs to make two separate requests with two separate RSL descriptions. Of course, it is desirable to combine requests, and that is one possible service that can be built into the high-level services layer. For two examples of services that can handle combined requests, or *co-reservations*, see Section 7.3.

- There are not distinct functions for distinct types of reservations, but instead there is a single function for each operation, such as cancel, for all types of reservations. These functions represent reservations with *reservation handles*. This reservation handle is an opaque string; that is, applications should not try to interpret the contents of the handle. Because it is a string, it can be easily saved to disk or passed to another program. Because all types of reservations use the same type of reservation handles, applications can build reservation strategies and code that are often independent of the type of reservation.

  After an application makes a reservation, it is provided with the reservation handle if the request succeeds, or an error if it does not succeed. In C-style code, this looks like:

  ```
  error = make_reservation(description, &handle);
  ```

  A reservation can be modified similarly:

  ```
  error = modify_reservation(handle, new-description);
  ```

  and cancelled similarly:

  ```
  error = cancel_reservation(handle);
  ```

  Notice that errors are reported directly to users when the function has executed.

  Similarly, a query can be made to discover the status of a reservation. Another important operation is the *bind* operation. This provides information about the reservation that is required to make it work, but the information is often not be known at the time a reservation is made. For example, a network reservation may be made long before the network connection is made, but GARA needs to know the TCP or UDP port numbers being used for the network connection in order to recognize packets that belong to the reservation. This information is provided with the bind operation.

- Resource managers can provide feedback to applications directly by calling a function (called a *callback*) in the application. Applications must request this directly, and they provide an appropriate function to be called by GARA. The arbitration layer is in charge of setting up the communication channel for the resource manager to communicate with the application.

  This callback can be used to provide simple feedback to the application, such as "the time for your reservation has just expired," as well as much more interesting feedback, such as "you are sending data at a faster rate than the reservation you made allows." Examples of interesting uses of the feedback provided through the callback are found in Section 7.2.

Security service

When a user begins communication with the arbitration layer, the first thing the arbitration layer does is to authenticate and authorize that the user is allowed to communicate with GARA.

This is an important aspect of GARA because not everyone is allowed to make a reservation for a resource. Administrators wish to control who has access to resources and such control requires authentication to ensure that the system knows who a user is and authorization to ensure that a user is allowed to make the reservation.

The authorization in GARA can range from simple authorization by consulting a list of authorized users, to advanced policies that restrict what time periods that a user is allowed to have a reservation for. Note that advanced policies may require knowing, for example, the start time of a reservation, but this cannot happen before the syntax/semantic check that happens below. Therefore, there may be some intermingling of the security service with the syntax and semantic checks.

Details of the actual mechanisms used for authentication and authorization used in GARA are found in Chapter 6.

Syntax and semantic checks

After the client has been authenticated and authorized, the arbitration layer will check that the syntax and semantics of the reservation request are valid. This step is the most important when a client is making or modifying a reservation, because this operation requires the most details from the client.

The syntax check is a straightforward check that the attribute-value pairs have valid attribute names and the values are specified correctly. For example, “lots” is not an acceptable value for the bandwidth, which requires a number.

The semantic check is also straightforward, in part because the arbitration layer can not fully check the semantics but must leave some of that to the resource manager. However, the semantic check can ensure that users do not specify reservations occurring entirely in the past, reservations that have negative durations or negative bandwidth, and other such anomalies. Additionally it can check that all of the required attributes are specified. For example, both start-time and duration must be specified when making a reservation.

The semantic check must also make sure that if a reservation handle is specified that it is actually a reservation handle. All client interactions except for the initial creation of a reservation specify a reservation handle.

Selection of resource manager

After the syntax and semantic checks, the arbitration layer selects the resource manager that will be handling the client’s request. Because each reservation is made separately, only one resource manager is needed. This procedure has two common cases:

- When a reservation is created, the client specifies what type of reservation is needed, such as a network bandwidth or disk space reservation. Because GARA only supports one underlying resource manager for a given type of reservation, the selection is a straightforward mapping. More complicated mappings are interesting to contemplate, but we do not envision that such mappings are likely to be needed.

- When a reservation is manipulated after it has been created, the same resource manager that was used to create the reservation must be used to manipulate it. Therefore the reservation handle encodes which resource manager created the reservation and the arbitration layer can conveniently select the same resource manager for future operations on the reservation.

Translation

Once the arbitration layer has selected the resource manager and parsed the description of the parameters, the parameters can be translated for the underlying resource manager. There are two types of translation that need to be performed:

- The reservation handle must be decomposed into a form that is recognizable by the resource manager. Different resource managers have different methods for referring to reservations. Whatever the resource manager expects is encoded in the reservation handle so that the arbitration layer can easily extract the necessary information.

- Functions such as creating and modifying a reservation may require specific attributes to be translated. For example, a user may request time on a CPU by specifying the percent of the CPU time they wish to use, but the underlying resource manager may expect a time duration within a time interval (such as 90ms out of every 100ms). Some translations of this sort are be easy to make, others are not so simple.

Communication with resource managers

Finally, the arbitration layer communicates with the resource manager and provides it with all of the parameters that have been checked and translated. The arbitration layer waits for a response, then translates that response into a standard GARA response. Different resource managers use different methods for reporting and labeling errors and for describing reservations. When a resource manager reports an error, the arbitration layer translates it into a GARA error and reports it directly back to the user through the GARA API. If a reservation was successfully made, the reservation description is translated into a GARA reservation handle that can be manipulated by the client.

Note that this communication with resource managers, while not conceptually complicated, is a core feature of GARA. There are a wide variety of resource managers: some were developed as part of the GARA implementation effort, and some were developed by other research groups. Because of this wide variety in resource managers, there are a wide variety of methods of communicating with these managers. This portion of the arbitration layer encapsulates all of these different communication methods, insulating the clients from the complication of the variety of communication.

4.3.3 Resource management layer

Beneath the arbitration layer is the resource management layer. For each resource for which we can make a reservation, there is a corresponding resource manager. This resource manager is responsible for several tasks:

- Accepting requests for reservations and performing admission control to decide if those requests can be accepted.
- Interacting with the underlying resource (such as the network or computer) in order to ensure that the reservations are guaranteed during the time interval over which they are valid.
- Publicizing information about the unused reservation capability so that clients can make decisions about what reservations are reasonable to request.
- Providing feedback to applications to inform them when their reservations begin and end, as well as notice that a reservation is too small. For example, a resource manager may inform an application that it is sending data faster than the reservation allows, and therefore the application may not see the performance it expects.

Several resource managers have been created as part of the GARA project. Because we wrote them, the task of the arbitration layer is simplified because the translation of user parameters is simple, and the method of communication with resource managers is similar between all GARA resource managers. On the other hand, several resource managers have been developed and integrated into GARA by other research groups, as described in

Section 7.1. These resource managers may require different types of interaction than the resource managers we develop, but the arbitration layer provides a uniform interface to all of the resource managers.

Resource managers are described in more detail in the next chapter.

4.4 Summary

The three-layer GARA architecture we have described here is a powerful architecture. The arbitration layer provides uniform access to a wide variety of underlying reservation mechanisms, and the high-level services layer builds additional useful services on top of the arbitration layer. Users can choose to interact with the high-level services layer for simplicity, or directly with the arbitration layer for have maximum flexibility.

4.5 Historical development of GARA and design decisions

It is often the case that the developers of a system such as GARA learn many things during the development process. Unfortunately, the lessons learned from the twists and turns of the development process are not always published.

When GARA was first developed [22], it attempted to provide more functionality than the current version of GARA. The original version aimed to provide a uniform interface not only to reservations, but also to manipulating the objects for which the reservations were made. For example, the object for a compute reservation is the process or set of processes that are running during the time of the reservation. The object for a network reservation is the network flow. GARA then provided parallel function calls for reservations and objects:

Reservations	Objects
create_reservation(description)	create_object(description, reservation)
modify_reservation(reservation)	modify_object(object, reservation)
cancel_reservation(reservation)	cancel_object(object, reservation, description)
...	...

The create_object function instantiated the processes or network flow. The cancel_object would kill the processes or close the network flow.

Our original feeling was that this was an elegant framework. But experience showed that it was too general for two reasons. First, it was hard to define the semantics of create_object for all types of reservations we could imagine. For example, what does it mean to create an object for a disk reservation? Creating a file is inappropriate: a program may simply want to write into a filesystem, or want to access pre-existing files with a guaranteed bandwidth. Similarly, what does it mean to create an object for a graphics pipeline reservation? (Reservations for graphics pipelines have been added to GARA by another research group, see Section 7.1.) We found that it was too difficult to shoehorn all the different functionality into a single create_object function.

Second, using create_object requires users to use a new interface to functionality they already have with other interfaces. For example, many programmers already have fully developed programs and libraries that use other mechanisms for submitting jobs to computers or for creating network flows and they wish to add QoS with as little effort as possible. It is easy for them to add support for making a network reservation, but difficult to retool their applications to use alternate methods for creating the network flows. Additionally, it is hard to provide the full range of functionality that programmers expect: when they create a network object, can they specify TCP buffer sizes and the entire range of socket options they expect on Unix? Do they have to use alternate access functions to send and receive data?

Because of these difficulties, we simplified GARA to provide just reservations and not objects. This change did not make GARA less general, but made it more useful. For example, as we describe in Section 7.4, it was easy to add network QoS capabilities to a high-level scientific communication library (MPI). Had we required MPI to create network flows with our own mechanisms, it would have caused a significant amount of additional effort, perhaps enough that it would not have been attempted.

CHAPTER 5
RESOURCE MANAGERS

5.1 Purpose of a resource manager

As we saw in Chapter 4, resource managers are the heart of GARA. They are responsible for handling all aspects of reservations including admission control, bookkeeping, monitoring, and interacting with the resource for which the reservations were made. For example, a resource manager may configure routers or CPU schedulers to enforce network or CPU QoS reservations respectively.

Figure 5.1 gives an abstract view of the functionality that a resource manager must provide. We will now look at each of these pieces of functionality. Realize that these are conceptual pieces, and in a particular resource manager these may be combined into single modules, or separated into more modules.

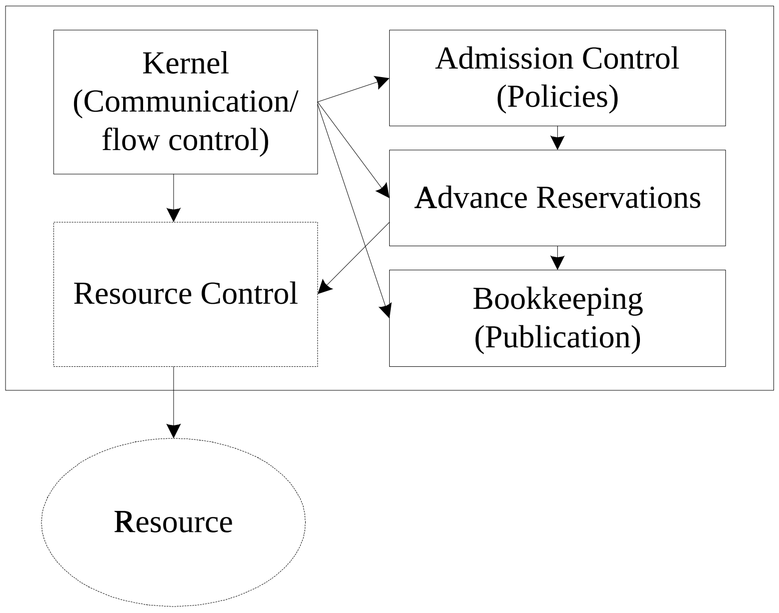
Kernel (Communication/ flow control)
Admission Control (Policies)
Advance Reservations
Resource Control
Bookkeeping (Publication)
Resource

Figure 5.1: An abstract picture of GARA's resource managers

5.1.1 Kernel

Each resource manager has a kernel which is responsible for accepting external communication, interpreting it, and deciding how to react. When the arbitration layer (described in Chapter 4) communicates with a resource manager, it is communicating with the kernel.

The kernel is responsible for providing feedback to applications via the arbitration layer. At a minimum, this feedback includes notification when a reservation begins or ends. Resource managers often provide additional feedback, such as notification that an application is oversubscribing its reservation. For example, an application may be sending data faster than its reservation allows, and it would be to the advantage of the application to either make a larger reservation or to slow down its rate of sending. To provide this feedback, the kernel interacts with the other subsystems within the resource manager. Examples of using feedback in interesting ways can be found in Section 7.2.

5.1.2 Admission control

When a request to make a reservation arrives, it is processed by admission control. Admission control is divided into two steps: first the resource manager decides if the reservation is reasonable and allowable. This is a policy decision. Second, it decides if there is capacity for the reservation.

The first step overlaps with the arbitration layer's security service, which implements some policy procedures to decide if a user is allowed to make a reservation. This is a necessary overlap for two reasons. First, the security service necessarily makes a decision without full knowledge of the internals of the underlying resource manager. For example, a resource manager may have a policy to only accept reservations from graduate students when more than fifty percent of the capacity of the resource is available. Such knowledge is not available to the arbitration layer, so it cannot support such policies. Second, some resource managers were written by other research groups, and they provide their own policy management independent of the policy management already in the arbitration layer.

5.1.3 Advance reservations

In order to support admission control, the resource manager must provide a subsystem that keeps track of advance definite reservations. This subsystem implements mechanisms for keeping track of all the reservations that have been made, and a decision procedure for deciding if a new reservation can fit with the existing reservations.

5.1.4 Bookkeeping and publication

In addition to the basic tracking of reservations, the resource manager must provide other bookkeeping. This includes:

- Permanent storage for advance reservations on disk, in order to recover from short-term failures such as a crashed computer or resource manager.
- Storage of extra information for each reservation. For example, while all reservations include the time that it begins and ends, network reservations also have information about the two computers involved in the reservation.
- Information about reservations that have already been made. This information is made available through a publication service and includes basic information about reservations so that entities wishing to make new reservations can make educated decisions about what times may be available for reservations. This information can be published in a number of ways, usually through an LDAP ([56]) directory. Other publication methods are described in Chapter 6.

5.1.5 Resource Control

Resource managers are responsible for controlling the resource for which reservations have been made. We must assume that resource managers have exclusive access to the resource, even though some resources may not be well controlled. Without this assumption of exclusivity, resource managers cannot actually guarantee any reservation, since another entity can configure resources in a way that interferes with the reservations made through the resource manager.

5.2 Implementation of resource managers

One reason to conceptually organize the design of resource managers as shown in Figure 5.1 is because each element is a candidate for code sharing. Most resource managers will need to implement their own kernels and resource control subsystems, but the other subsystems can often be implemented once as libraries and shared among resource managers. As described in Chapter 6, this is how several resource managers were implemented in GARA.

Such code sharing is not generally available when using resource managers developed by different authors. In fact, there is a wide range of resource manager functionality among different resource managers, and not all of them may implement the functionality required by GARA. Externally written resource managers can be integrated into GARA in two different ways:

- Some resource managers happen to implement the full functionality needed, and the arbitration layer handles interaction with such resource managers just like any other.
- Some resource managers implement a subset of the functionality of GARA. Most commonly, resource managers implement immediate reservations but not advance reservations. In this case, a *wrapper* resource manager can be created. This resource manager provides the additional functionality that is necessary and communicates with the underlying resource manager. While this is a workable technique (indeed, we will see an example of such a wrapper resource manager in Chapter 6), the wrapper resource manager needs to assume that it has exclusive access to the underlying resource manager, just as resource managers assume exclusive access to the resource. Essentially, the underlying resource manager becomes a resource for the wrapper resource manager.

5.3 Design of a resource manager

Section 5.1 described resource managers in terms of the functional blocks they must implement. However, the functional blocks were fairly abstract, and not much detail was provided. This was appropriate because of the wide variety of resource managers that can be implemented in widely varying manners.

In this section, we look at an alternative way of describing the behavior of resource managers in terms of sensors, actuators, and decision procedures (from the language of [46]). This description will become particularly relevant in Section 7.2 when we describe how these three mechanisms can interact to provide adaptation. For now, we limit ourselves to describing what these mechanisms are.

Because sensors, actuators, and decision procedures describe resource managers at a finer level of detail than the functional blocks we used earlier, there is a greater variety from resource manager to resource manager. One example of how these mechanisms may combine can be seen in Figure 5.2.

The three mechanisms are:

- *actuators* that permit online control of resource allocations or application behavior;
- *sensors* that permit monitoring of resource allocations or application behavior; and
- *decision procedures* that allow entities to respond to sensor information by invoking actuators.

As illustrated in Figure 5.2, these three elements act in concert to achieve adaptive control. For example, in a network resource manager a sensor might signal a nonzero loss rate associated with a flow at a router, which indicates that an application is sending data too quickly. (As described in Section 6.5.1, when a user sends data too quickly, it is dropped.) A decision procedure in the associated application can then execute to determine whether to reduce the sending rate or alternatively, generate a request to a resource manager to increase the reservation for that flow by invoking an actuator.

Although the discussion that follows emphasizes the importance of these three elements for adaptive control, they are equally important for describing the behavior of resource managers.

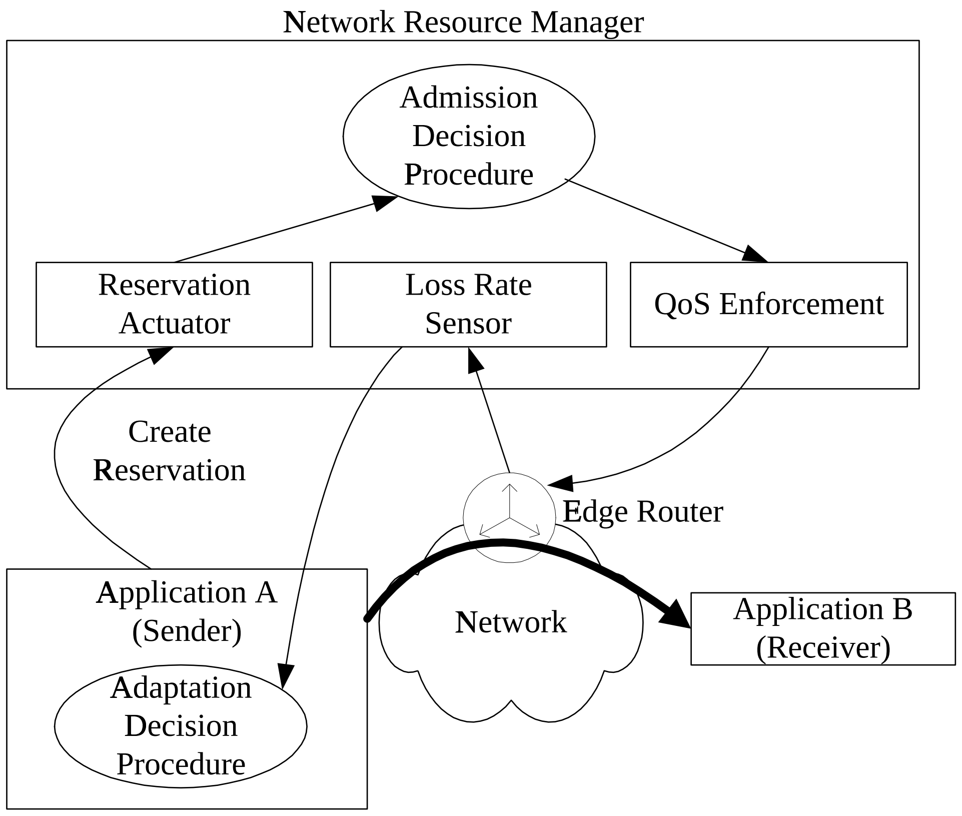

Figure 5.2: An example of how actuators, sensors, and decision procedures may be combined within a resource manager. This particular example shows a network resource manager.

5.3.1 *Actuators: online control*

A first prerequisite for adaptation is support for online control of resource characteristics. Resource managers fulfill this requirement directly via control functions that allow an application to make and subsequently modify QoS reservations, as described in Chapter 4. GARA examines and translates the application's requests using the arbitration layer before passing it to the resource manager's reservation actuator. This reservation actuator uses an admission control procedure to decide if the reservation can be accepted, and the actuator communicates the answer back to the application via GARA.

5.3.2 Sensors

A second requirement for adaptive control is that we are able to determine the state of system components and detect state changes. Resource managers contain the functionality in sensors, and can provide the sensor's input to the application through the callback mechanism described in Chapter 4. We have implemented two such sensors in our GARA prototype.

All resource managers implement a reservation change sensor. This sensor reports when a reservation begins or ends. This sensor can be used in two ways. First, it is part of the publication mechanism described above, to inform entities seeking information about the state of reservations in the resource manager. Second, applications do not have to closely monitor the time in order to know when a reservation begins or ends: they are notified when it begins. This is particularly useful when the application cannot rely on having its local clock precisely synchronized with that of the resource manager.

Some resource managers implement other sensors as well. For example, a network resource manager may monitor how many packets that belong to a reservation are being dropped because they are being sent over the reserved bandwidth. (See Chapter 6 for more information about how our network resource manager works.) This sensor therefore detects that an application has made an incorrect reservation that needs to be adjusted. An example of this is described more fully in Section 7.2.

5.3.3 Decision procedures

The third component of an adaptive control architecture comprises the decision procedures that invoke actuators in response to sensor data.

In our environment, such decision procedures can occur in multiple locations: not only do they appear in resource managers, but also in applications. This enables interesting adaptive behavior, because both the resource manager and the application are able to react to changing conditions.

Decision procedures may be invoked within a resource manager at a number of points. Following authentication, an incoming request is first authorized and then executed. Decision procedures may be invoked at both stages: for example, to determine whether a request should be granted, in the first instance, and to reallocate resources in the second instance if the newly authorized reservation oversubscribes available resources.

A particularly interesting decision procedure is our network resource manager's bulk-data transfer decision procedure, which is described in Section 7.2.

CHAPTER 6
IMPLEMENTATION

Chapters 4 and 5 described the GARA architecture. We now provide a detailed description of the GARA implementation and motivations for the decisions made in the implementation.

6.1 Implementation overview

Figure 6.1 shows an overview of the GARA implementation. This should be compared to Figure 4.2 on page 19, the overview of the GARA design. The arbitration layer in Figure 4.2 is not contained in a single spot within our implementation, but is spread between two processes: the gatekeeper and the GARA service, both described below. In turn, the GARA service uses the Local Reservation and Allocation Manager (LRAM) API, which contains the final pieces of the arbitration layer.

The reason for this division is partly practical and partly historical. GARA was developed within the context of the Globus project [20], and the gatekeeper is a standard Globus component. Because we were able to leverage off of this existing infrastructure we not only developed a prototype of GARA more quickly, but the infrastructure simplified distribution of GARA to external developers. Although this was not of theoretical importance, it was of great practical importance: system administrators that have already installed Globus (which is widely used) trust the security mechanisms that GARA uses because they are the same security mechanisms used by Globus.

GARA is implemented in C, and works on a variety of Unixes.

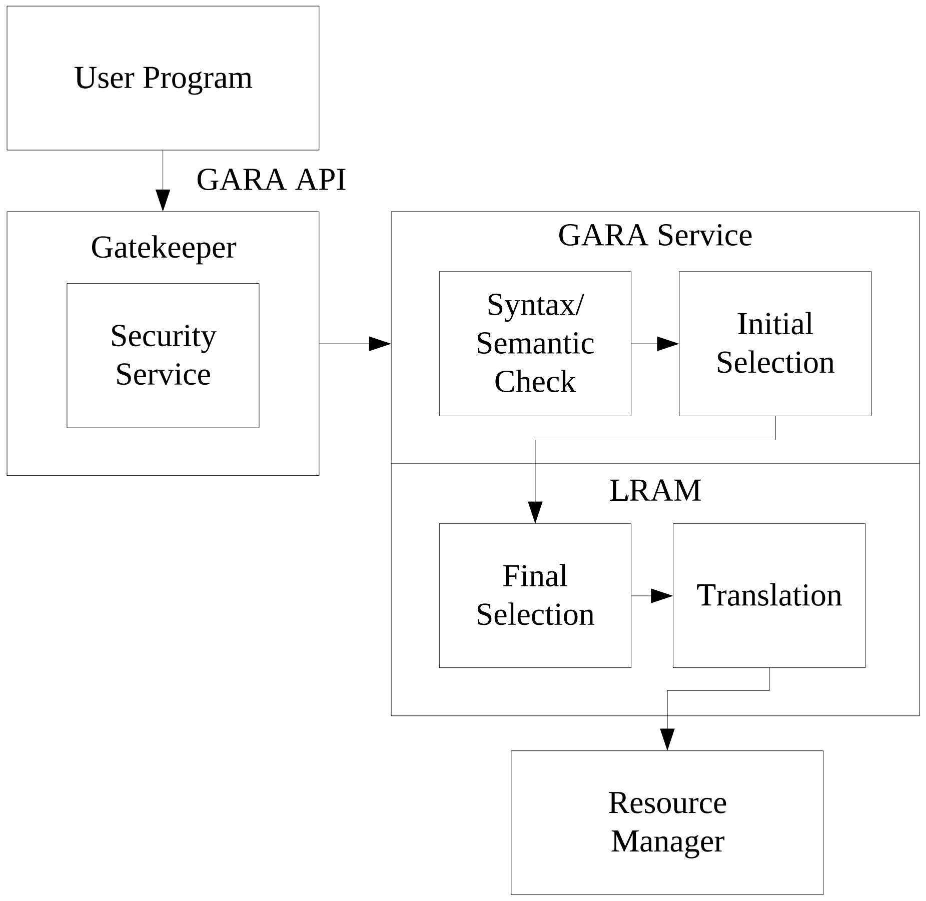

Figure 6.1: An overview of how GARA was implemented. See text for details.

6.2 Client interaction with GARA

Clients communicate with the arbitration layer by using the GARA API. This API is contained within a library that is linked into an application. The API is described briefly in Chapter 4, and in detail in Appendix A.

The GARA API uses standard Unix socket functions for network communication to the arbitration layer. For each function call that requires communication with a resource manager, the API opens a new connection to the gatekeeper and provides the identity and credentials of the user so that the gatekeeper can authenticate and authorize the user. The identity and credentials are provided by Globus [23], which uses the Generic Security Service API (GSSAPI) [36] to provide user credentials and authentication. While the GSSAPI can theoretically provide a variety of security mechanisms to Globus, in practice Globus normally uses either X509 certificates [7] and PKI authentication, or Kerberos [52].

The client communicates all parameters over the connection that is opened, and the arbitration layer communicates reservation handles and any errors using the same connection.

A good candidate for future optimization would be to reuse connections to the gatekeeper in order to avoid the overhead of the repeated authentication for a single reservation. However, since we expect that most reservations exist for a long time and are infrequently modified, this optimization was not a priority.

6.3 Gatekeeper

The gatekeeper authenticates a user, as described in Section 6.2, not only for GARA, but for other services such as job creation. Because the gatekeeper normally runs as root on Unix systems, system administrators prefer that the gatekeeper is as simple as possible. Therefore the gatekeeper does not provide services such as QoS or job creation directly, but after a user has been authenticated and authorized, the gatekeeper launches an external program to handle the request. This external program is called a gatekeeper service or in our case, the GARA service.

When a user connects to the gatekeeper, it must specify which service it wishes to use. Once the user has been authenticated, the gatekeeper consults a lists of users that can be authorized to use services on the machine. For each user that is listed, the services available to that user are listed. This means that a user can be authorized to use GARA, but not another service controlled by the gatekeeper. This is the extent of authorization performed by the gatekeeper. More sophisticated policy-based authorization such as “graduate students can only make reservations for times between 8:00pm and 5:00am” cannot be performed by the gatekeeper but must be performed either by the GARA service (described next) or within a resource manager. This is because the gatekeeper merely passes the user’s request to the gatekeeper service and performs no interpretation of the request.

6.4 GARA service

As described in the previous section, the GARA service is a process created by the gatekeeper after the user has been authenticated and authorized, and it implements the portions of the arbitration layer other than the security service.

The GARA service uses the Local Reservation and Allocation Manager (LRAM) library and API to communicate with resource managers. Originally, it was thought that applications wishing to make reservations locally could also use the LRAM API and avoid the overhead of the remote connection and security. This has not turned out to be useful. However, the LRAM API is useful for another reason: after the GARA service parses the users request, it uses the LRAM API as a partially abstract interface to resource managers. That is, the GARA service can parse and check users requests, but it does not need to understand the details of communicating with the resource managers. This simplifies construction of the GARA service, which usually does not need to be modified for new resource managers, it just needs to be recompiled with a new LRAM library.

The LRAM API is not as abstract as the GARA API. Unlike the GARA API, which only has a single function for making a reservation, the LRAM API has a set of functions for making reservations, one for each type of underlying resource: network, CPU, etc.

The LRAM library is responsible for the translation of parameters from the parameters provided by the user to the parameters used by the particular resource manager.

6.5 Resource managers

In Section 5.2, we described two different varieties of resource managers: those that implement the full functionality needed by GARA, and those that need a wrapper resource manager to provide additional functionality. In this section, we give examples of these types of resource managers that were implemented in GARA. However, we will break down the resource managers into three types:

1. *Native resource managers* were developed entirely within the development of the GARA development effort. Because of this, they integrate easily with the rest of

GARA. They also allow us to directly experiment with different types of QoS. An example of a native resource manager is the network QoS manager that we built.

2. *Full-functionality third-party resource managers* were developed entirely by other groups and they provide all the functionality that GARA expects. For this type of resource manager, we only had to modify the LRAM library to provide translation and communication with the resource managers. An example is the Portable Batch System (PBS) job scheduling resource manager which is interfaced to through the LRAM library.

3. *Wrapper resource managers* were developed within the context of the GARA development in order to provide additional functionality to externally developed resource managers that do not provide all of the functionality that is expected by GARA. An example is our wrapper for the process QoS resource manager, which uses the Dynamic Soft Real Time scheduler (DSRT). DSRT provides immediate reservations, so we built a wrapper resource manager that adds advance reservations and callbacks.

We will look at each of these resource managers in turn.

6.5.1 Network QoS resource manager

We built a resource manager from scratch to provide network QoS using differentiated services. Before explaining the highlights of the resource manager, we will take a brief look at *differentiated services*, also known as *diffserv*. Further reading about diffserv is also available in [44, 2, 33, 30, 43].

Differentiated services

Early work in network QoS, such as Tenet [15] and RSVP [4] with Integrated Services [3] required that each router between the sender and receiver recognize and individually treat packets belonging to each reservation separately. This is expensive since each router must devote memory and processing power to recognize and treat each reservation.

In the late 1990's, researchers became concerned with this heavyweight methodology. While it can provide excellent QoS for a small number of data streams, it is not clear whether it scales well to the entire Internet. If reservations become widely used across the Internet, then core routers, those routers in the heavily used backbones of the Internet, could conceivably have to recognize and process millions of simultaneous reservations. This would require a considerable amount of memory and processing power, increasing the expense and complexity of the core routers. Some researchers even questioned if routers could be made that can support so many reservations at the high speeds that are necessary in the Internet backbone.

Therefore, researchers began proposing an alternative method of providing network QoS called differentiated services, or diffserv [44]. The basic idea of diffserv is that we push the hard work of network QoS to the edges of the network and keep the work in the interior of the network as simple as possible. To make the discussion that follows clear, we will divide routers into two types: edge routers and core routers. A router is on the edge if it has an interface through which a computer or non-routed network is directly connected. A router is a core router if it has an interface that connects to another router. Note that, with this definition, a router may be both an edge router and a core router.

In diffserv, computers or edge routers recognize packets entering the network that belong to reservations and mark those packets as belonging to a traffic aggregate. That is, packets from different reservations are marked with the same marking, making them part of the same traffic aggregate. Core routers do not recognize individual reservations, but do recognize traffic aggregates based on the simple markings in the packets, and they treat different aggregates differently. For example, an edge router might recognize all traffic from my computer and mark it as priority traffic, another edge router might recognize video traffic and mark it as priority traffic, and the core routers might recognize priority traffic and give it preferred treatment. Figure 6.2 illustrates this.

At first glance, it may not be obvious how this methodology can provide us with QoS reservations for individual flows since we are operating on aggregates of flows. In fact, some service providers will not choose to use diffserv to provide individual reservations, but will provide broad classes of service. For example, a provider may guarantee that Gold traffic gets better service than Silver traffic, which gets better service than Bronze traffic,

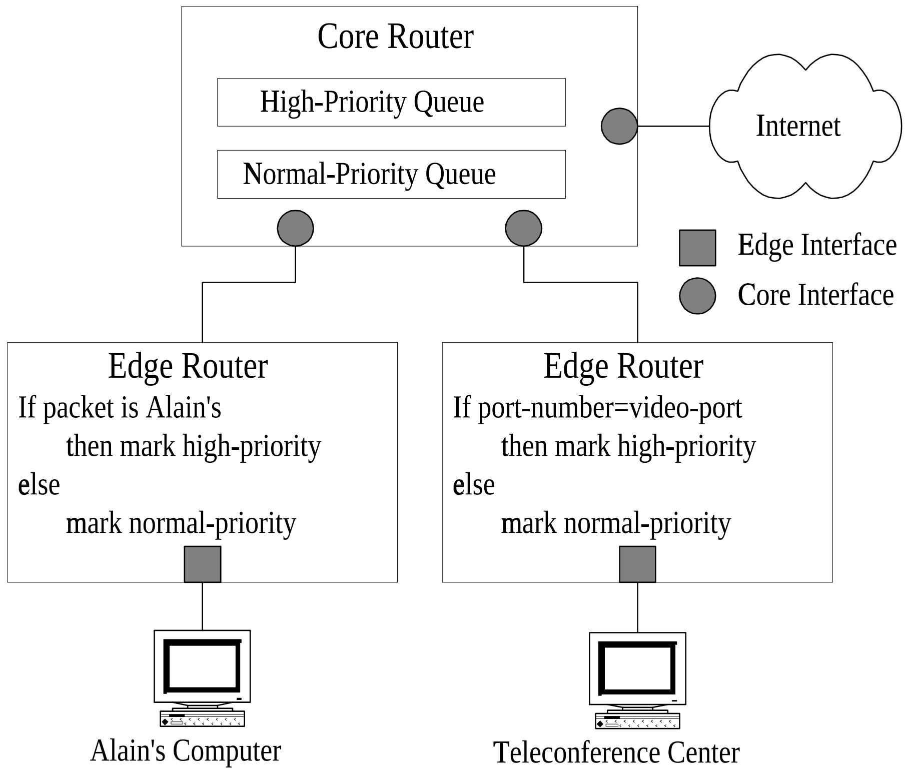

Figure 6.2: An example of differentiated services. Two edge routers each recognize certain kinds of packets as being important and mark them as priority traffic. The core router can then treat these marked packets as high priority without recognizing particular QoS reservations

and users sign up for a particular class of service for all of their traffic. However, with careful admission control at the edges of the network and careful provisioning in the core of the network, it is possible to provide QoS reservations with diffserv, as described below.

The marking in a packet is called the Differentiated Services Code Point (DSCP). A DSCP indicates a particular treatment that a router should provide. This treatment is known as a Per-Hop Behavior (PHB). Note that this treatment is not a global or end-to-end behavior, but a treatment at a single router. If we want end-to-end services, we need to build them out of PHBs. Currently the Internet Engineering Task Force (IETF) has defined two types of PHBs: Expedited Forwarding and Assured Forwarding.

Expedited forwarding Packets marked with the Expedited Forwarding (EF) PHB can be intuitively thought of as being assured that they will be transmitted as quickly as possible, with as little delay as possible (and therefore as little jitter as possible) and with no loss of packets, as long as the EF packets do not exceed a certain rate. Packets that do exceed the configured maximum rate are dropped, so as not to interfere with other traffic.

EF is defined more precisely as:

> The EF PHB is defined as a forwarding treatment for a particular diffserv aggregate where the departure rate of the aggregate's packets from any diffserv node must equal or exceed a configurable rate. The EF traffic *should* receive this rate independent of the intensity of any other traffic attempting to transit the node. It *should* average at least the configured rate when measured over any time interval equal to or longer than the time it takes to send an output link MTU sized packet at the configured rate. [33]

However it should be noted that although the IETF published the RFC describing the EF PHB on the standards track, discussion continues as this dissertation is being written about what the actual definition of EF should be, and the final definition will be slightly different than the current definition. However, if you keep the above intuitive description in mind, you will have a good feeling for how EF should behave.

EF can be implemented in a straightforward way using Priority Queuing (PQ). PQ allows a single class of packets in a router to always be sent before any other packets. That is, there is a strict two-level priority, and the priority queue will always be empty before the best-effort queue is serviced. (PQ can also be implemented with other methods such as Weighted Fair Queuing, and even Weighted Random Early Detection, as demonstrated in our earlier work.)

It should be clear that if we use PQ and assign only EF packets to the priority queue that our packets will be sent as described above: they will be sent as quickly as possible with as little delay as possible. Assuming we configure our routers carefully, they will also not be dropped. However, there is one important concern with PQ: it is possible to starve the best-effort traffic if there is sufficient priority traffic.

Dealing with starvation is also straightforward if careful admission control is done: reservations should only be accepted if it will not create more EF traffic than can be handled and traffic should be policed in the edge routers to ensure that no application sends more EF traffic than it has made a reservation for.

Assured forwarding Assured Forwarding (AF) is a more complicated PHB than EF. AF defines four classes of service. There is no ranking of the classes, but the service provider defines how much of a router's resources each class gets. This allows a service provider to provide their own unique services to users, but makes it harder to build end-to-end services that go through multiple domains.

Within each class, packets are assigned one of three different drop probabilities. This is a way of marking packets that are more or less important so that if packets need to be discarded, the less important packets are discarded first. For example in an MPEG movie I frames are more important than P or B frames, so the P frames should be marked with a lower drop probability and the P and B frames can be marked with a higher drop probability.

While it is possible to build higher-level network reservations using AF, it is harder than with EF, because there is no clear definition of what the different classes mean, and different domains might define the different classes differently. Therefore, it is hard to build an end-to-end reservation service using AF. For this reason, in the experiments that follow, we exclusively use the EF PHB. This is not to suggest that building end-to-end reservation services with AF is impossible or not worthwhile, but rather that we chose to focus our efforts on EF.

Managing network reservations

In GARA, a network resource manager provides admission control and network configuration for a single network domain. The definition of a domain is not exact, but is a policy decision. Generally the network domain will correspond to an administrative network domain. For example, a resource manager may control QoS within a University's network.

When the resource manager receives a request for a reservation that goes through its domain, it decides if there is sufficient bandwidth in the domain to allow the reservation.

When a reservation is granted, it will automatically configure the appropriate edge router to classify traffic belonging to the reservation, police that traffic to ensure that it stays within the limits of the reservation, and mark the traffic with the DSCP that corresponds to EF.

While the demonstrations that follow occur within a single network domain, it is possible to use GARA when a reservation needs to be made across multiple domains as shown in Section 7.3.

Network QoS testbed

The network QoS resource manager was tested in a local-area testbed called the GARA Advance Reservation Network Testbed (GARNET) at Argonne National Laboratory. (In work with a collaborator, GARA was also tested in a wide-area testbed [51].) This testbed can be seen in Figure 6.3.

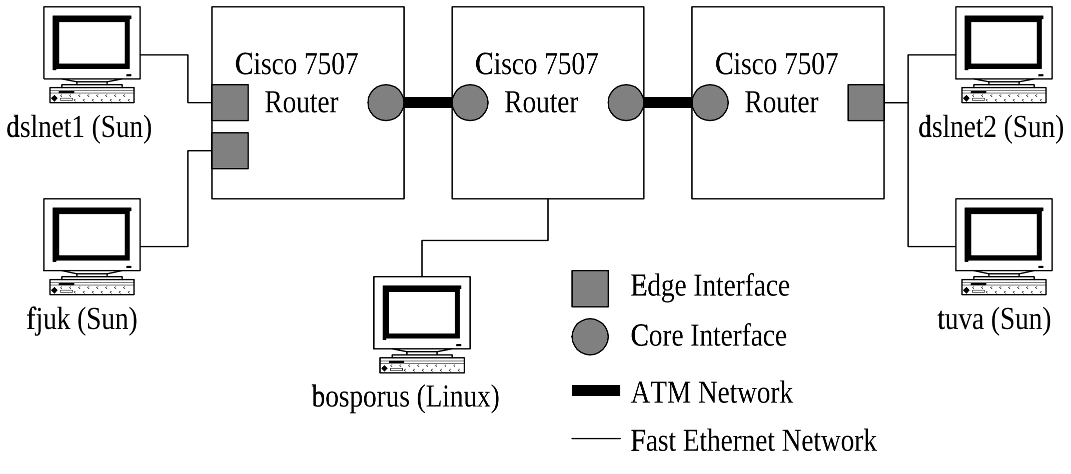

Figure 6.3: The local-area GARNET Testbed.

All of the demonstrations that follow were performed on this testbed. The network is composed of three Cisco 7507 routers and a large number of computers including four Sun workstations, several Linux computers, and a multi-processor SGI computer. Note that there were more computers attached at either end of the testbed than are shown in Figure 6.3. Also note that the edge router connected to dslnet2 has two Fast Ethernet interfaces while the edge router connected to dslnet1 has only one Fast Ethernet interface.

While it would be more convenient to have a second interface, one of the interfaces failed and could not be replaced.

The Cisco routers provide mechanisms we used to implement the network QoS:

- *Classification of traffic* To classify traffic, we used Cisco's Modular QoS Command-line interface (MQC), which allows an entire spectrum of packet classification from loose specifications such as "all packets on this interface" to tight specifications such as "UDP packets being sent from address/port to address/port". GARA classifies packets tightly, as in the second example. MQC also marks the packets with the appropriate DSCP to indicate that packets should receive EF treatment.

- *Policing* To police traffic, we also used Cisco's MQC. MQC allows traffic to be policed at a specific bit rate, although it has to be an even multiple of 8 Kb/sec. Traffic is allowed to have a burst rate using a token bucket mechanism, which allows short periods of an increased bit rate. This prevents applications which do not have perfect control over sending rate from losing packets when they occasionally send too fast.

- *Shaping* Although it is not used in any of the demonstrations below, MQC also allows traffic to be shaped. This can be useful for applications that have difficulty controlling their speed so that they do not have packets dropped. Applications that use TCP may have particular difficulties with this, since TCP will often send traffic in bursts at line speed. A close collaborator has done experiments with GARA and shaping, described in [50].

- *Queuing* The Cisco routers provide priority queuing that can be selectively applied to different classes of traffic. Our routers are used to provide priority queuing to traffic that has been marked as EF. The total amount of bandwidth that can be given to priority traffic is configured in advance, not at runtime, by the system administrator.

Note that although we have used Cisco routers and mechanisms specific to these routers, there is no intrinsic requirement to use them. In fact, a collaborator has modified the router configuration portion of GARA (which is conveniently located in a script that is separate from the main portion of the resource manager program) to configure Extreme routers from Juniper Networks [32]. Extreme routers have capabilities similar to those of the Cisco router, but are configured differently. However, it was straightforward to modify the network resource manager to work with the Extreme routers.

Demonstration of network QoS

With this testbed, we have demonstrated our ability to provide network QoS. Although many of the detailed results have been documented elsewhere ([25, 51, 50, 24]), we provide an overview here.

In the demonstrations that follow, we used TCP and UDP traffic generators capable of generating traffic at a constant rate. These traffic generators were used both for the traffic that had reservations and the traffic that attempted to interfere with the reserved traffic by causing congestion. The congestion traffic was always UDP traffic because UDP does not slow down in response to congestion, unlike TCP, thereby allowing us to maintain a controlled rate of congestion on the network.

UDP Figure 6.4 demonstrates a simple experiment demonstrating network QoS under congestion. In this experiment, traffic was sent from fjuk to tuva (see Figure 6.3), while congestion was sent from dslnet2 to bosporus. The router-to-router ATM links were configured at 50 Mb/s to make them easier to fully congest. The application began sending data and was able to meet its target speed of 20 Mb/s until congestion began. Shortly thereafter, a reservation was made with GARA for the application, and the application was again able to sustain its desired rate, until the reservation was canceled. At the end of the experiment, the congestion was stopped.

Note that the application was not able to have a perfectly steady 20 Mb/s even when there was no congestion and no policing enabled. This was due simply to the difficulty in maintaining a high-speed but perfectly steady rate.

Figure 6.5 shows a similar experiment. In this case, there were 5 simultaneous streams of traffic, each of a different rate. The start of the reservations were staggered because the router was not particularly responsive to simultaneous attempts to configure it while

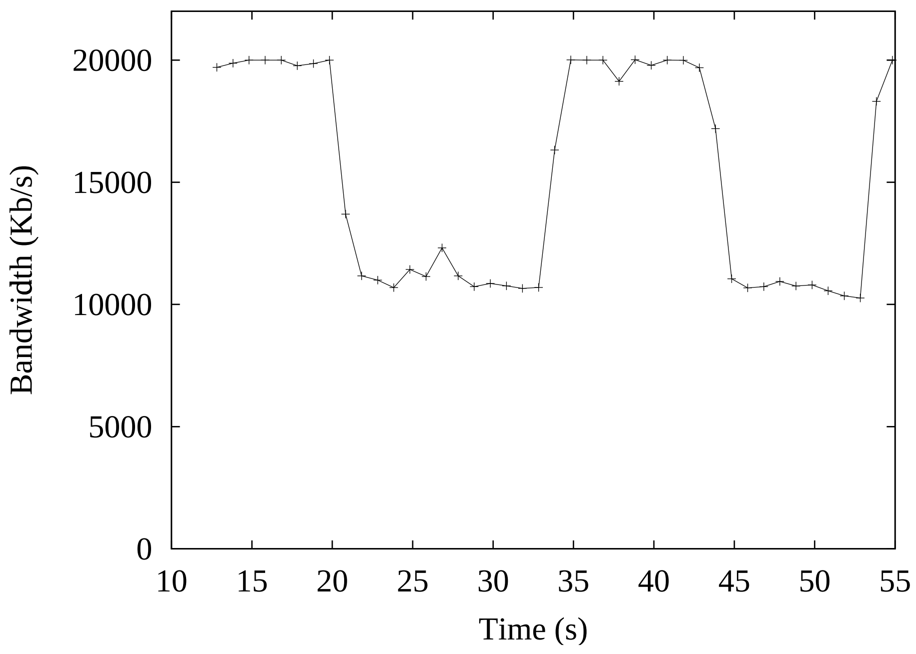

Figure 6.4: This graph shows the throughput an application attempting to send 20 Mb/s of UDP traffic was able to receive. At about time 21, congestion began and the application slowed. At about time 34, the application made a reservation and was able to sustain its data rate although the congestion continued.

traffic was extremely heavy. However, it can be seen that the routers were able to maintain simultaneous reservations while there was heavy congestion.

This example was particularly interesting because it did not involve five pairs of computers but was done between a single pair of computers, so the streams of traffic were competing for the same resources on the same machine. While the machine can clearly handle it, it did affect the precise timing so that the applications did not receive perfectly steady data rates.

TCP Providing network QoS to applications that use TCP is a tricky task. We have found, in personal discussions with other researchers, that many people do not believe that TCP is an appropriate protocol to use with network QoS. This is because TCP implements congestion control mechanisms that slow down its transmission. When packets are lost in networks, it normally indicates that there is congestion in the network, and TCP assumes

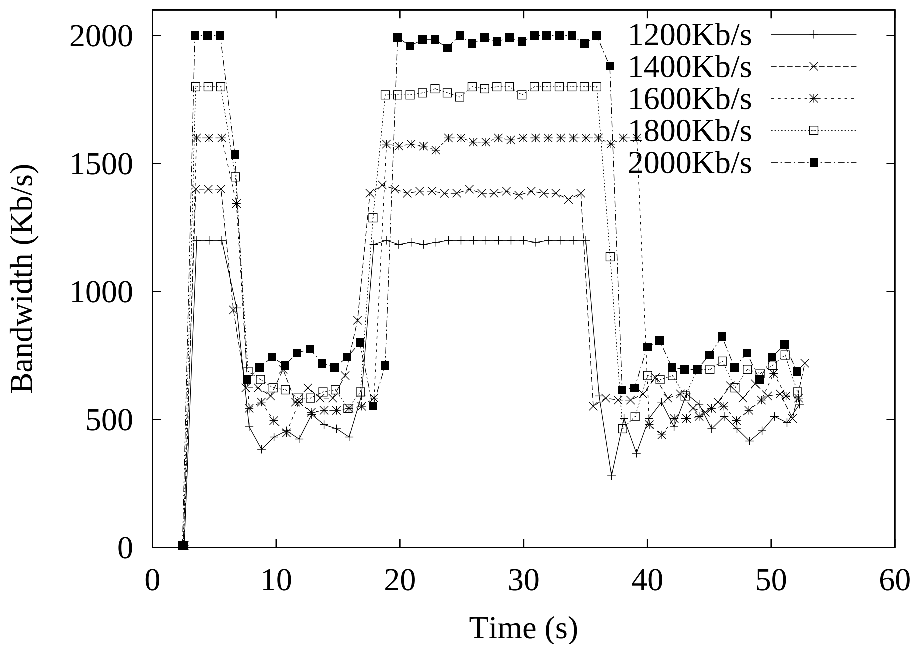

Figure 6.5: Reservations for five simulatenous reservations for UDP traffic. See text for details.

that lost packets indicate that it should slow down in order to be a good neighbor and help to eliminate congestion.

When using network QoS such as the expedited forwarding service provided by GARA, lost packets do not indicate congestion, but indicate that the application is sending data too fast for its reservation. However, TCP does not react differently to packets dropped by the policing mechanism and it slows down. In some cases, several packets in a row may be dropped, causing TCP to go into slow-start congestion control, which slows TCP down significantly below the reservation rate.

This can be seen in Figure 6.6, which shows a TCP application attempting to send data at a steady rate of 20,000 Kb/s. We made three different reservations, to show the effects of policing at different rates. There are two important lessons to be learned from this figure. First, it is insufficient to make a reservation for the desired application rate, because it does not take into account the packet overheads. Second, a reservation that is too small may cause the TCP rate to fluctuate greatly.

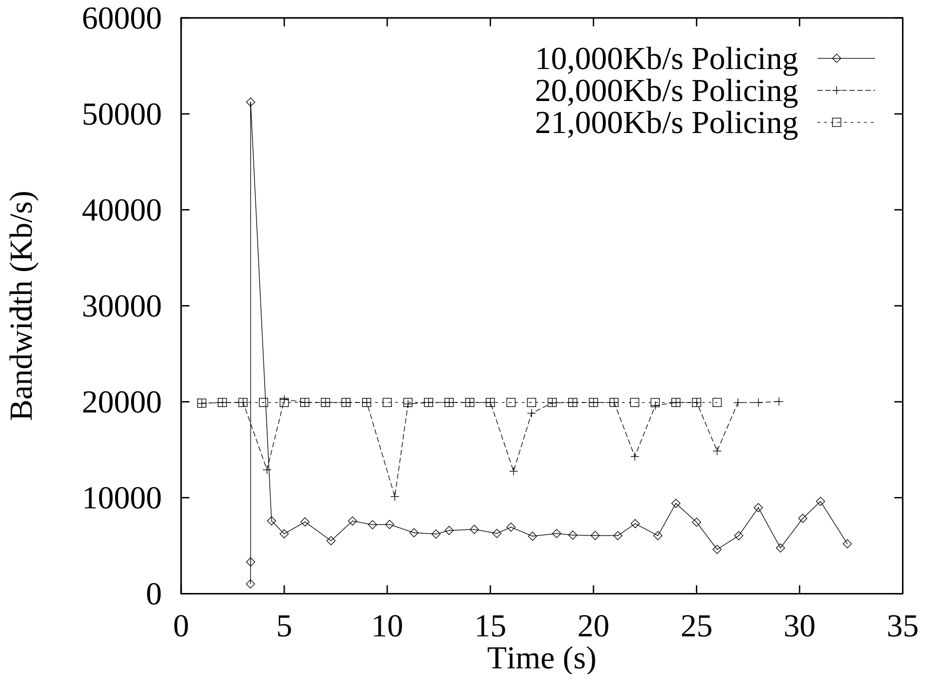

Figure 6.6: The effects of under-reservation on a TCP stream. The application is sending at 20,000 Kb/s. Notice that when the reservation is for 10,000 Kb/s, the application receives 6,000 to 7,000 Kb/s. This is due to TCP's congestion control.

However, making a reservation that is large enough does work for a TCP reservation, as can be seen in Figure 6.7. In this figure, we have a TCP stream sending at a rate of 20 Mb/s. Around time 6, heavy congestion began (caused by the UDP traffic generator). Immediately, TCP's slow-start congestion mechanism forces TCP to slow down transmission to about one-sixth of the desired sending rate. A reservation began about time 16 and lasted until about time 25. During this time, the TCP application was able to achieve its desired traffic rate. After time 33, the congestion stopped, and the application was able to operate at full speed with no reservation.

Figure 6.8 shows five simultaneous TCP streams. Congestion begins at about time 8 and the reservations begin in a staggered fashion after time 20. This figure shows that everything "mostly works". All of the streams were sent from a single source (fjuk, in Figure 6.3) to a single destination (tuva). Not only was there competition between the TCP streams and the UDP congestion, but the streams were competing on the source and

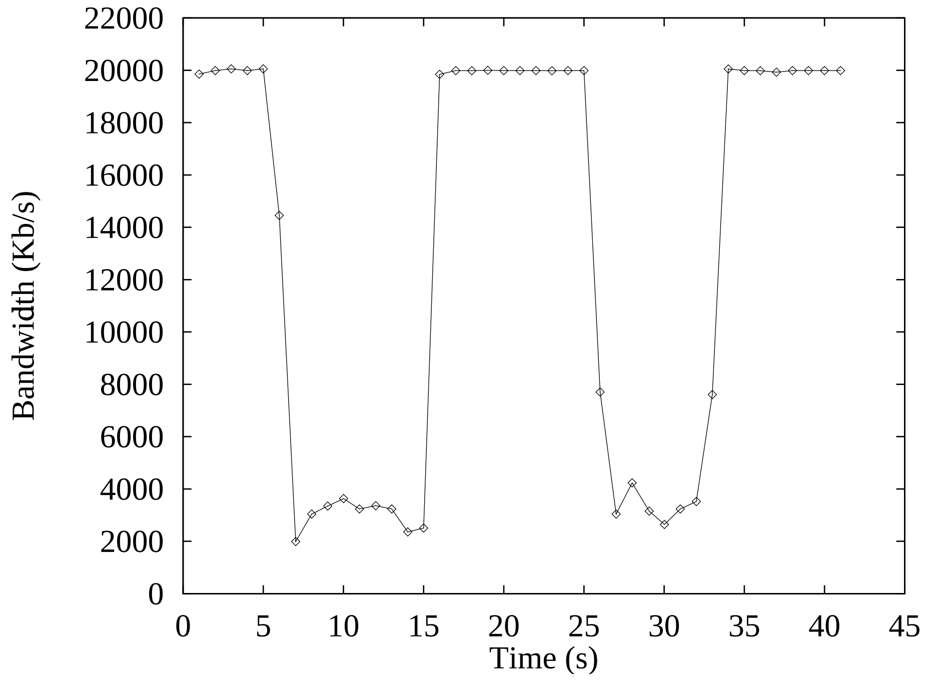

Figure 6.7: Demonstration of a network reservation for a TCP flow. Congestion began at time 6 and end at time 33. A reservation was in place from time 16 to 25. See text for details.

destination computers as well as the ingress and egress Ethernet interfaces on the testbed. In any case, the reserved traffic shows performance as good as the performance when there was no congestion. (Other than a single low point in the 2,000 Kb/s TCP stream.)

These results demonstrate that it is practical to provide QoS to programs that use TCP. This is particularly important because many existing programs use TCP and users wish to have QoS without rewriting the programs.

6.5.2 PBS resource manager

The Portable Batch System (PBS) resource manager was written independently of GARA. PBS schedules batch jobs submitted to parallel computers or clusters of computers. Originally, PBS had no notion of reservations, and it would have been somewhat difficult to connect with GARA, although it could have been implemented similarly to the DSRT wrapper resource manager described below. A recent version of PBS added support for advance

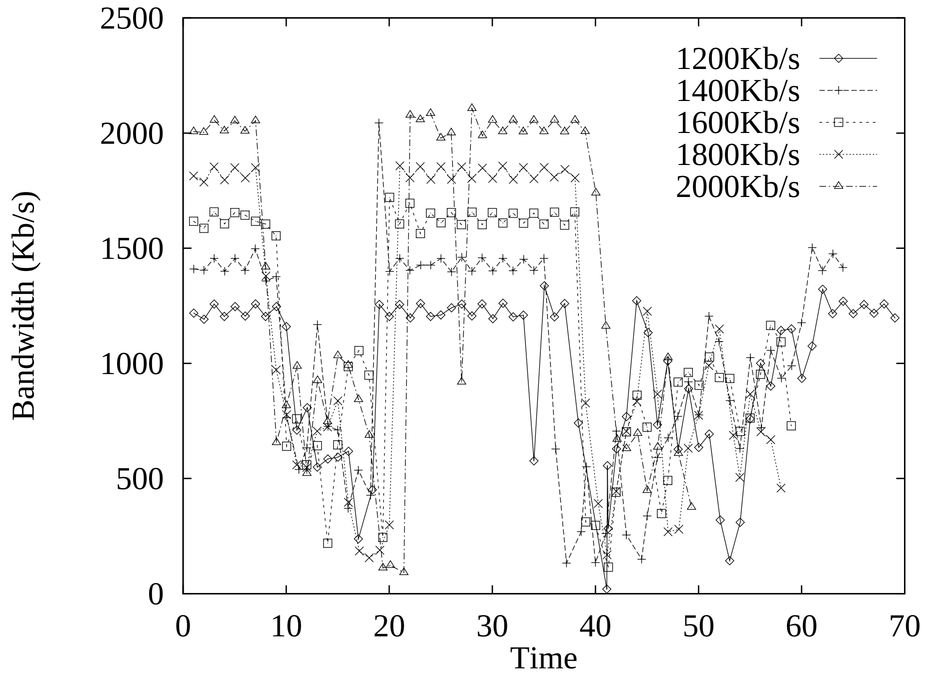

Figure 6.8: Demonstration of a network reservation for five simultaneous TCP flows. Congestion began at time 8 and lasted until time 60. Reservations began in a staggered fashion after time 20 and listed for about 15 seconds each. See text for details.

definite reservations, and it was straightforward to use GARA to make reservations using PBS.

This was done by modifying the LRAM library—we did not need to build any additional resource managers. In this case, the fact that the LRAM is provided as a library became particularly useful: the GARA service needed little modification, while the LRAM library encapsulated the communication with PBS.

When a user requests an advance reservation from PBS both the start time and duration are specified, as in GARA. PBS decides whether or not the reservation can be granted based on the other advance reservations already granted and the maximum running times of all the jobs it currently has. All jobs have a maximum running time, and if a job exceeds that time, it is cancelled. Therefore, PBS can decide whether or not advance reservations can be granted.

If a reservation is granted, PBS creates a queue. This queue will be active during the reservation time and inactive at all other times. Users can submit as many jobs as they desire to the queue, and they will be run normally, as long as the queue is active. When the reservation ends, the queue will be removed, and any of the jobs that were running from the queue will be removed.

PBS uses this queue name as an identifier for the reservation. Therefore, the LRAM library encapsulates the PBS identifiers within the GARA reservation handle. Because the GARA reservation handles are flexible and opaque to users, this was not a difficulty.

At the SC2000 conference, we demonstrated the advance reservations using GARA, Globus' GRAM service for running jobs, and PBS by making reservations for two computers at geographically remote locations. (See Figure 6.9.) We then submitted a job to both computers before the reservation started. The jobs waited in the queue until the reservation time began, then they communicated with each other using the Message Passing Interface (MPI). Note that this is an example of co-reservation, which is discussed in Section 7.3.

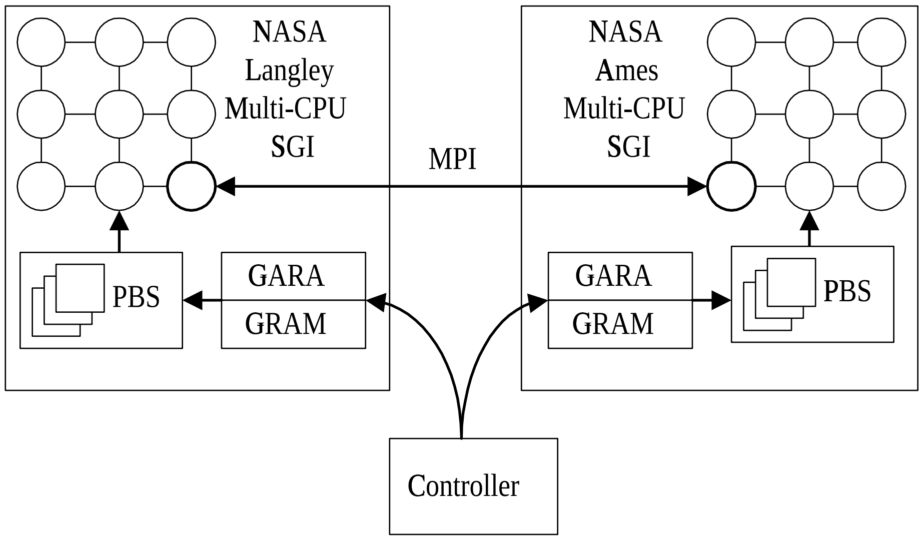

Figure 6.9: At the SC 2000 conference, we demonstrated GARA using PBS and advance reservations with the setup in this figure. NASA Langley is in Virgnia while NASA Ames is in California. Advance reservations were made before the jobs were submitted and the jobs communicated using MPI.

6.5.3 DSRT resource manager

The Dynamic Soft Real Time Scheduler (DSRT) is a user-level process that runs at a higher priority than the Unix scheduler and takes over the work of scheduling in order to provide soft real-time guarantees to applications. DSRT was built by a research group at the University of Illinois Urbana-Champagne and is described in detail in [9].

As described above, DSRT allows applications to make only immediate indefinite reservations for scheduling. (It also allows immediate definite reservations to be made for very short periods.) The reservations it provides are flexible: reservations can be made for a periodic guaranteed time slice, for variable time slices specified with burst parameters, and for adaptive reservations. Adaptive reservations allow applications that do not know how much time they need to make their best guess, then receive help from DSRT in determining the reservation that meets their needs.

To implement advance reservations on top of DSRT, we built a resource manager that communicates with the DSRT scheduler using the DSRT API (see Figure 6.10). We then perform admission control and reservation tracking just as we do for network reservations. When a GARA reservation begins, we make an immediate reservation with DSRT. When a GARA reservation ends, we cancel the immediate reservation with DSRT. We have to assume that GARA is the only program that is allowed to make immediate reservations. With that assumption, this scheme works well. In fact, with the current implementation other programs can make reservations "behind our back" with DSRT, but if we were interested in making a bullet proof implementation, we could easily modify DSRT to allow only GARA to make reservations. The best way to do this would be to implement authentication and authorization using the mechanisms already available in Globus. Alternatively, we could modify the DSRT scheduler to allow only one connection to the DSRT scheduler, and the DSRT resource manager would make that connection.

While DSRT allows very flexible reservations, such as "I need 90ms out of every 150ms", we chose to implement a simpler reservation scheme, that allows users to request a percentage of the CPU times. Our hope is that this sort of reservation can be easily translated to other real-time schedulers with absolutely no change for users. We translate a reservation of 85% of the CPU, for example, to be .085 seconds every .1 seconds. Other

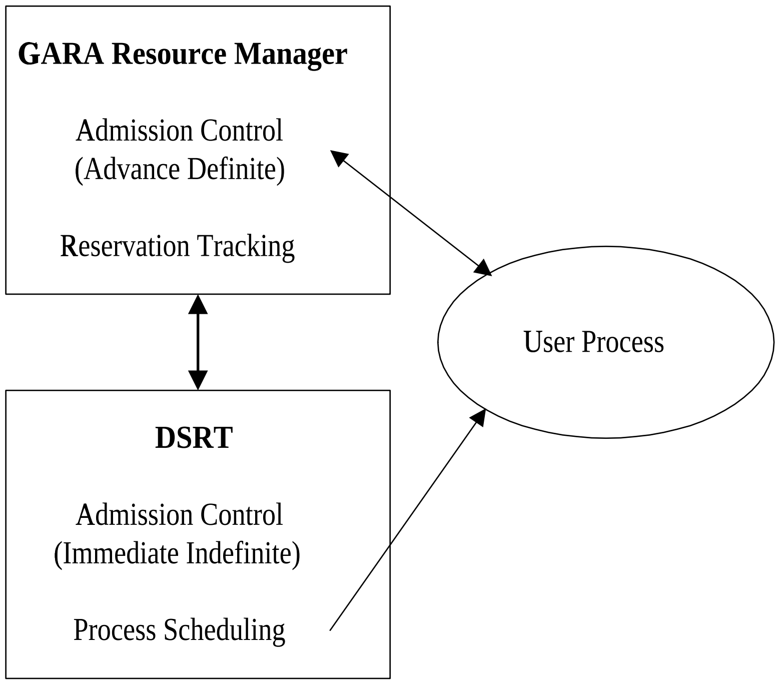

Figure 6.10: GARA works with DSRT by providing advance definite reservations on top of DSRT. User processes make reservations with GARA instead of DSRT.

translations might be more or less effective depending on the application. It would not be hard to extend the GARA reservation request to allow a finer granularity in the reservation. For instance, instead of merely specifying a reservation description that contains:

```
&...(percent-cpu=85)...
```

it could optionally contain an extra field, cpu-interval:

```
&...(percent-cpu=85)
    (cpu-interval=.1)...
```

This is likely to be fairly portable, and provide extra flexibility to the applications that could benefit from it.

This implementation demonstrates the flexibility of the GARA architecture. We have taken an underlying reservation system that allows only immediate indefinite reservations and built an advance reservation system out of it. Also, DSRT is only accessible from Java and C++ using DSRT-specific interfaces, while GARA can make reservations using DSRT with the same interface used with other underlying reservation systems.

Figure 6.11 shows an example of using DSRT. This figure shows an application that is sending a TCP stream at 80 Mb/s. At time 40, the application has made a network reservation, and the competing traffic is not affecting it at all. However, at time 59, the application is not able to send at its full rate because another application has begun using a large portion of the CPU. At time 81 the bandwidth mostly returns to the full rate because the application made a CPU reservation using GARA. (Actually, the reservation was made by an external program. As we noted in Chapter 4, GARA is flexible about who actually makes the reservation, and we took advantage of that flexibility here.) In our experience, DSRT is not a perfect scheduler, but merely gave us “pretty good” results. This is due in part to the fact that DSRT cannot prevent interrupts from affecting the time that an application can receive. Nevertheless, the bandwidth with CPU reservation is significantly better than without it.

Note that the particular “CPU hog” that competed for the CPU was a simple program doing meaningless trigonometric math, but it could have well been another program. This particular CPU hog was a consistent CPU hog, while other programs might only periodically hog the CPU, so this makes for a clearer demonstration. However, CPU QoS is equally important whether the CPU competition is occasional or consistent.

6.5.4 Other resource managers

In addition to the resource managers described above, we developed several other bare-bones resource managers, mostly to demonstrate the ease with which we could extend GARA. In particular, we implemented:

- *Priority CPU Resource Manager* This resource manager allows reservations to be made for a certain priority level. Only a single process may have a reservation at a time.

- *Exclusive CPU Resource Manager* This resource manager was closely integrated with the Globus job creation utility, GRAM. Any number of non-priority processes

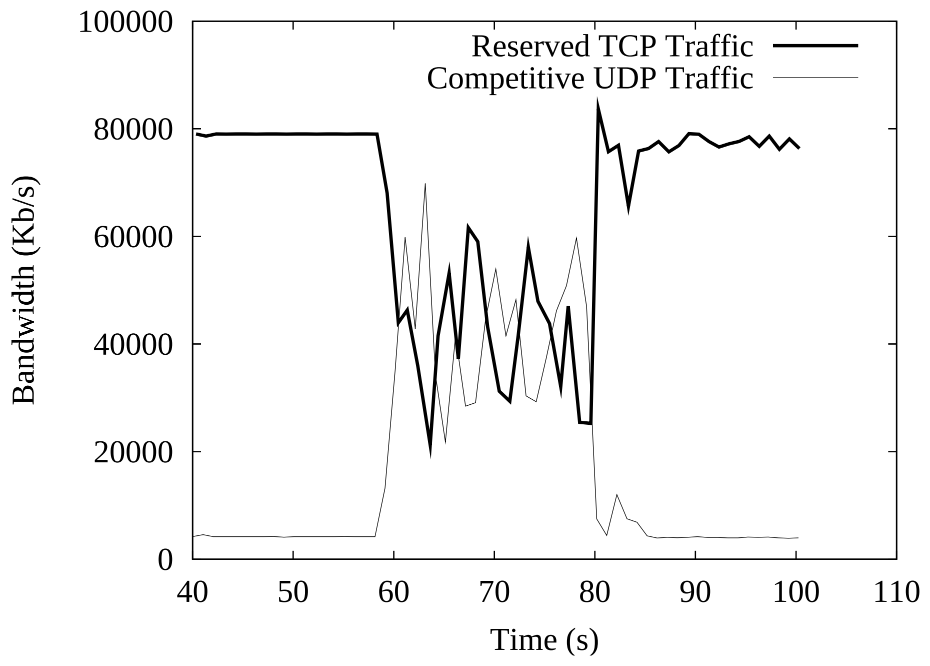

Figure 6.11: An example of using DSRT. At time 40, the application is sending a TCP stream at 80 Mb/s with a network reservation. At time 59, competition for the CPU begins and affects the performance of the TCP application until a CPU reservation begins at time 81.

are allowed to run simultaneously, unless a job with a reservation is running, in which case all other processes are killed. That is, when a process with a reservation begins, all other user processes are killed. This was used for a demo in which users wanted to run lots of processes at a time, except during emergency times, when a single process should have full access to a computer.

- *Disk Space Resource Manager* This resource manager allowed users to make reservations for disk space. Disk space was allocated using the quota system on a single controlled disk. A better implementation would use a logical volume manager, because the quota system does not allow the resource manager to clearly distinguish between multiple reservations for a single user, while a logical volume manager would.

- *DPSS Resource Manager* The Distributed-Parallel Storage System (DPSS) [55] provides a high-performance, distributed, network-accessible data cache that can be shared by many different users. It is usually used to provide access to very large data files at very high speeds. To provide high speeds, several computers with several disk controllers work together to provide the illusion of a single high-speed block device that is accessible across a network.

Our resource manager allows applications to make reservations for exclusive access to the DPSS server. While this does not guarantee any particular amount of bandwidth, this does allow users to be confident that the DPSS server will be responsive to their needs.

6.5.5 Notes on resource manager functionality

For the resource managers that we fully implemented, we were able to share a large amount of code between the different resource managers. In particular, we implemented a *slot table* for working with advance reservations. The slot table keeps track of all the advance reservations in a linked list, and had a decision procedure to check if a new reservation request can be admitted. The slot table works with a variety of reservation types because it allows resource managers to specify the capacity of the reservation as a floating point number without imposing an interpretation on the capacity.

In addition, other portions of the resource managers were shared, such as the communication libraries and the publication methods. For the publication methods, we allowed information about the current reservations (without the users' identity) to be published in three different methods: to a human readable file, a web page, or a machine-interpretable LDAP directory entry. Because of the method of implementing the publications, it is easy to add new methods of publishing information about reservations should the current methods not be sufficient.

CHAPTER 7
VERIFICATION

In Section 4.2, we listed some of the main features of GARA. We believe that GARA is an architecture that provides users with a great deal of power and flexibility. In particular, we claim:

1. GARA's modular design means that new QoS mechanisms are easily integrated into GARA.
2. GARA's feedback mechanisms provide a base for extensions that can significantly extend GARA's capabilities.
3. The GARA API makes it straightforward to develop higher-level functionality, including reusable, multi-purpose services.
4. GARA mechanisms are sufficiently abstract that they can be incorporated easily into higher-level programming models.

We will now look at each of these claims in turn.

7.1 Modular design

Claim: GARA's modular design means that new QoS mechanisms are easily integrated into GARA.

The best evidence for GARA's modular design is a set of extensions to GARA that show new QoS mechanisms that integrate easily. Since new QoS mechanisms are encapsulated within resource managers, we will demonstrate GARA's modular design by discussing resource managers that demonstrate GARA's flexibility.

As we discussed in Section 5.2, there are two ways to integrate resource managers. There are resource managers that provide exactly what is needed, or one can create a wrapper resource manager that adds additional functionality to the resource manager. In either case, the LRAM layer (described in Section 6.4) hides the details of communication with the resource managers, allowing much of the arbitration layer to be unmodified. Once the LRAM layer provides access to a resource manager the user can easily access the resource manager through the standard GARA API.

Four different resource managers demonstrate the ease of integrating new QoS mechanisms into GARA:

1. The PBS scheduler (Section 6.5.2) was easily integrated into GARA by modifying the LRAM implementation to communicate with PBS directly.

2. The Dynamic Soft Real-Time CPU Scheduler (DSRT) was integrated by developing a wrapper resource manager, as described in Section 6.5.3.

3. An alternative network resource manager was integrated into GARA by a research group at LBNL [32]. While this network resource manager also used differentiated services, it was an independent effort as part of the STARS project [31]. This resource manager supplied everything that GARA requires a resource manager to implement, although the internal mechanisms were often different than those used in the GARA network resource manager described in Section 6.5.1. Unlike GARA, it was implemented in C++ instead of C. However, this was transparent to most of the arbitration layer, because it was encapsulated in the LRAM implementation. The integration into GARA was about one day of effort.

4. A resource manager was developed to provide advance reservations for graphic pipelines on SGI machines [13]. Some high-end SGI machines provide multiple graphic pipelines which can be used locally or remotely to render graphics. Being able to make reservations for a graphic pipeline can ensure that the pipeline can be used by an application when it needs it. Once this resource manager was developed, it was an easy matter to integrate it into the rest of GARA, again making just minimal changes to the LRAM implementation.

These four resource managers demonstrate the variety of ways that resource managers can be integrated into GARA. This variety encourages us to believe that a wide variety of QoS types could be easily integrated. Indeed, in addition to the resource managers already described, we have prototyped other resource managers, such as disk space and remote disk bandwidth resource managers, as described in Section 6.5.4.

In all of these cases, the majority of work to integrate these new resource managers into GARA was in changing the gatekeeper service to parse user parameters. Due to an inflexible design, this was more complicated than it should have been. However, we believe that a recoding of the gatekeeper service would significantly simplify this process without affecting the rest of GARA.

We should note that, as flexible as GARA is, there are resource managers that do not integrate as easily into GARA. For example, when a resource manager requires users to use non-standard methods of access to resources, these resource managers do not benefit much from integration into GARA. An example would be a disk bandwidth reservation system such as Fellini [37], which requires users to use non-standard I/O functions to access disks with reservations. While Fellini could be integrated into GARA, it would not mesh well: users would use GARA to make reservations, but would be required to use Fellini specific functions to access disk. If the underlying disk reservation system switched from Fellini to an alternate system, users would be required to change their applications, which is something that GARA attempts to avoid.

Although this is an important example because it shows the limits of GARA, we believe that QoS mechanisms can be integrated into GARA without significant difficulty. These QoS mechanisms can be easily used singly or in concert by GARA users.

7.2 Feedback mechanisms

Claim: GARA's feedback mechanisms provide a base for extensions that can significantly extend GARA's capabilities.

In Chapter 4 we described GARA's callback mechanism which is used to provide immediate feedback about reservations to users. All resource managers are required to provide basic feedback about reservations: when a reservation is beginning and when it is ending.

However, some resource managers are capable of giving additional feedback. This feedback comes from sensors in resource managers, as described in Section 5.3.2, and can be fed into decision procedures (Section 5.3.3) to adapt the behavior of an application. We have built two different adaptation methods based on GARA's feedback to demonstrate the utility of GARA's feedback mechanisms.

GARA's feedback mechanism is particularly flexible because the arbitration layer, described in Chapter 4, melds the uniform abstract feedback interface provided to applications with the resource-manager specific feedback. Resource managers do not need to interact with users directly, but can focus on collecting sensor information that needs to be provided to the application and passing it to the arbitration layer, which will handle the interaction with the user.

Feedback to users is provided by two numbers. The first number is a constant describing the type of feedback. The second, optional, number is additional data for the feedback. Currently, there are four different types of feedback that GARA provides, and each one has a constant identifying the type. The actual number that is provided is irrelevant, so we just show the types below.

Type of Feedback	Parameter
Reservation beginning	None
Reservation ending	None
Network reservation too small	Percentage packets dropped
Change data rate	New data rate

The following two sections describe how we use the latter two types of feedback. These demonstrations show the utility in having a general feedback mechanism in GARA, and suggest the range of possibilities that it enables.

7.2.1 Learning bandwidth in applications

We first describe how adaptive techniques based on feedback can be used to determine the bandwidth reservation required to support a particular UDP flow. The motivation for this is that many application developers have no knowledge of QoS mechanisms or of the principles by which QoS parameters are determined. We show that feedback provided by a simple packet loss rate sensor can be used to guide a decision procedure that sets bandwidth reservations adaptively, increasing reservations until loss rates reach zero. This decision procedure can be incorporated in an application or in a separate agent. More information about decision procedures and sensors can be found in Section 5.3.

Our decision procedure uses feedback provided by a packet loss rate sensor. This sensor is part of the network resource manager described in Section 6.5.1. Periodically (usually every ten seconds) the sensor queries the routers that police reserved traffic. Network flows that send data too fast are policed and packets are dropped to ensure that they do not send data over the reserved limit. The sensor can query the fraction of packets dropped, P, and calculate $1 - P$ which is the fraction of packets that conformed to the reservation. Our decision procedure calculates what reservation would have been needed to ensure that no packets would have been dropped, as follows, where R_o is the old reservation and R_n is the new reservation.

$$R_n(1 - P) = R_o \tag{7.1}$$

or

$$R_n = \frac{R_o}{1 - P} \tag{7.2}$$

To evaluate the effectiveness of this strategy, we performed experiments as follows. In order to obtain a replicable experiment, we used as our application a test program that sends UDP traffic at a user-specified rate across our testbed. The program and testbed are the same as described in Section 6.5.1.

Results for two similar experiments are superimposed in Figure 7.1. In each case, the application made an initial reservation for 2500 kilobytes per second (KB/s) but then sent data at a higher rate: in the first case at 4000 KB/s and in the second case at 8000 KB/s. As described in Section 6.5.1, the first router classified, policed, and marked traffic. Because the router allows small bursts, the application initially was able to send slightly faster than the reservation allowed, but then the data rate settled down to a constant 2500 KB/s.

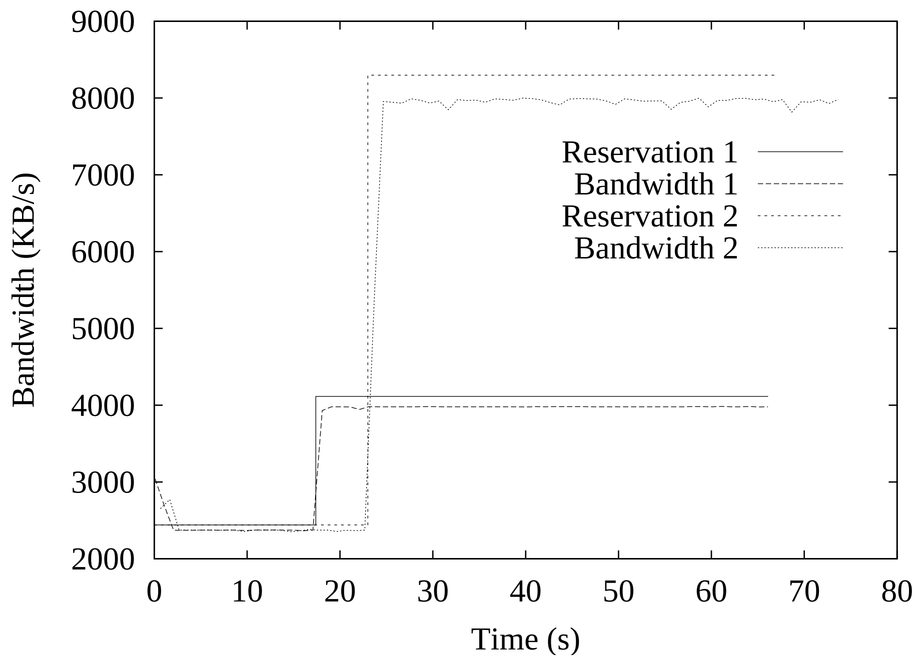

Figure 7.1: Here are two UDP flows that made reservations that were too small, but they were able to adapt their reservations upon receiving feedback about packet loss from the network resource manager. The bandwidth is less than the reservation because of packet header overheads.

Our loss rate sensor is implemented by our network resource manager, which queries the first-hop router every ten seconds and provides feedback to the application for every query except the first. (The first query is not reported to the application because we wish to gather statistically sufficient data.) Because the resource manager and application are not synchronized in any way, the feedback arrives at slightly different times in the two cases in the figure: at 16 seconds and 22 seconds, respectively.

The UDP application was able to adapt quickly in these experiments. However, because our router only updates packet drop statistics every ten seconds, the adaptation need not always work so well. For example, if the router statistics were gathered just as a series of packets were starting to be dropped, a unrepresentative result may be reported to the application. However, this problem would be compensated for after another round of adaptation.

It is possible to adapt the rate of TCP flows in reaction to the same loss information.

However it is complicated by the fact that TCP does not send at a steady rate due to congestion control mechanisms (see Section 6.5.1). Since applications cannot easily know the internal state of TCP, it is hard to calculate a new reservation using Equation 7.2. One possible solution is to use binary search for a reservation, as illustrated in Figure 7.2.

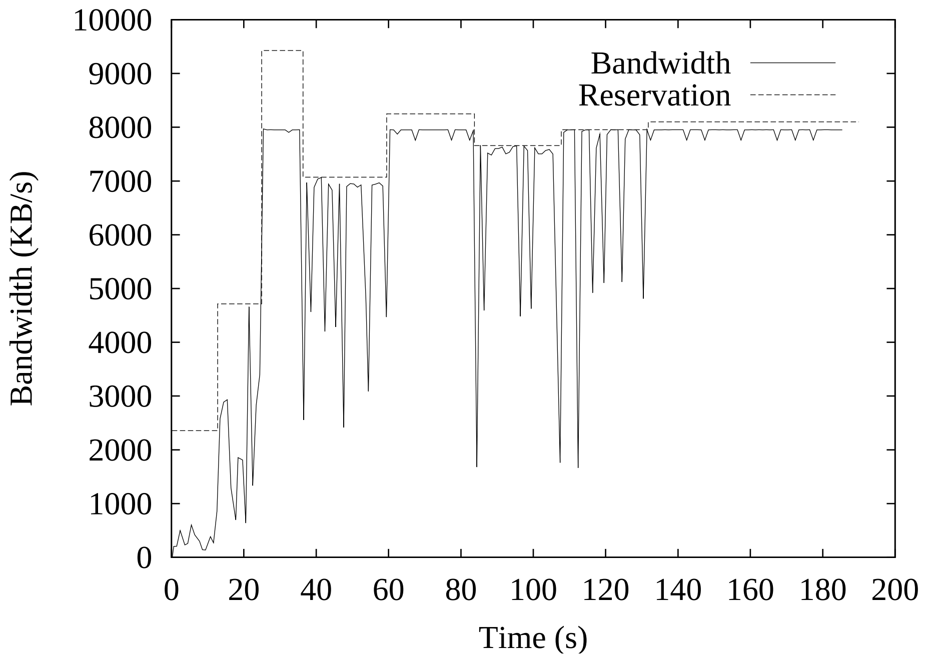

Figure 7.2: Here we see a TCP flow that uses binary search to discover the correct reservation for to use.

While it is likely that we can significantly improve upon the speed of finding a correct reservation in the TCP case, the important point is that the ability to adapt is because GARA provides the appropriate feedback mechanism. The packet loss sensor works with GARA's feedback mechanism and allows different types of applications (UDP- and TCP-based) to adapt their behavior.

7.2.2 Bulk data transfers

We experimented with an additional interesting service, built on top of differentiated services, called bulk-transfer. There are times when people need to transfer data quickly, but not at a particular rate. For instance, they may need to transfer a 10 TB file “by tomorrow, 6:00AM”, but they do not care about the actual transfer rate at a particular time, as long as the file transfer is completed in time.

Our bulk-transfer service is a substrate on which one could build such a file transfer service. It allows a reservation to be made for all of the bandwidth that is not currently in use by other reservations, up to the maximum allowed for premium reserved traffic. However this does not prevent other new reservations from being made but, as other, non bulk-transfer reservations are made, the amount assigned the bulk-transfer goes down accordingly. When the other reservations are no longer active, the amount is given back to the bulk-transfer reservation. The bulk-transfer reservation is always kept informed of the total bandwidth assigned to it using the callback mechanism described in Sections 4.3.2 and A.1.7.

A simple demonstration of the bulk-transfer service working can be seen in Figure 7.3. In the figure, the bulk-transfer initially makes a request for bandwidth and receives all of the 40 Mb/s available, and it is able to send data using TCP at a constant 40 Mb/s even though congestion begins at time 10. Three other reservations are made in succession, at times 23, 75, and 128. During each of these reservations, amount provided to the bulk-transfer application is decreased. When the application is informed of the decrease, it adapts to the new bandwidth. Note that when it adapts to the decrease, the applications transfer rate briefly drops more than it should. This effect occurs because the application is adapting the frequency at which it writes to the TCP socket buffers at the same time that TCP is adapting and reacting to a few packets dropped when the policing limit is decreased.

Figure 7.4 shows the results when this experiment was repeated between our testbed and a testbed at Lawrence Berkeley Labs in California. Notice that the result is similar, except that it takes longer for the application to adapt, particularly in one case. The particularly slow adaptation between time 40 and 48 was due to a combination of a bad device in the network and the larger TCP windows that are needed to sustain high performance in wide area networks do not allow adaptation to occur as quickly.

Our current implementation of bulk-transfer is a bit simplistic, and it needs two improvements to become a useful service. First, we currently only support one bulk transfer flow per domain. It is a straightforward extension to add multiple bulk transfers by simply sharing the bandwidth equally between all of the bulk transfers. Second, bulk transfers

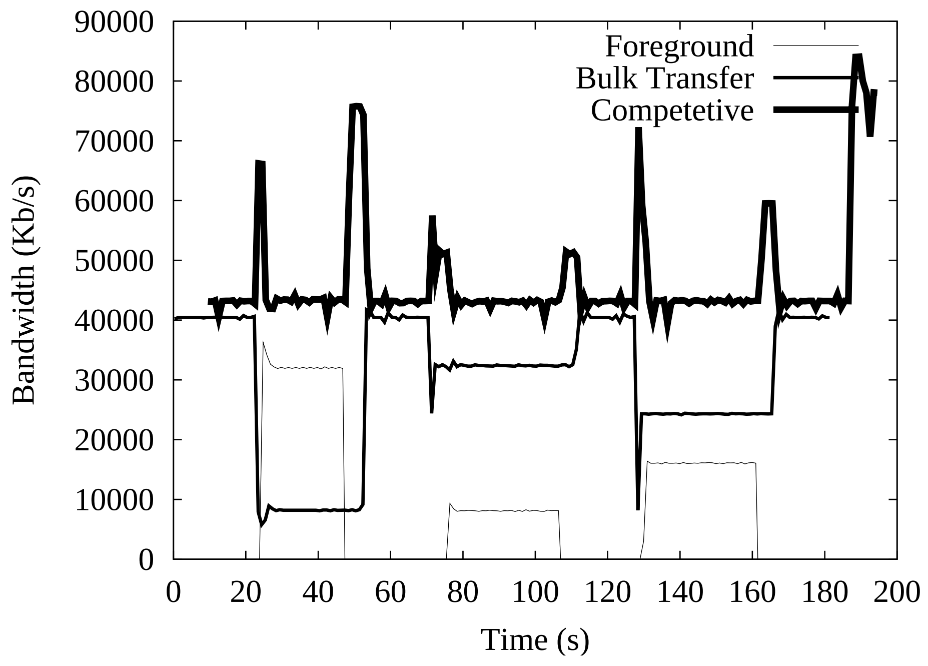

Figure 7.3: Bulk transfer in our local area testbed. See text for details.

should be able to specify a total amount of data that needs to be transferred, and a deadline by which time they need to transfer that data. If a total amount of data and a deadline are specified, one can derive a minimum bandwidth needed to fulfill the deadline. We can then calculate a proportional share for each bulk transfer flow that ensure that this minimum bandwidth is satisfied.

Assume that there are n bulk transfer flows, with minimum bandwidths $b_0, \ldots, b_{n-1}$. At any given time, there is unused bandwidth U, and we know that

$$U >= \sum_{i=0}^{i<n} b_i$$

This is because we do admission control to ensure that each bulk transfer flow will always have its minimum bandwidth. We will assign each actual bandwidth, B_i as:

$$B_i = U \cdot \frac{b_i}{\sum_{j=0}^{j<n} b_j}$$

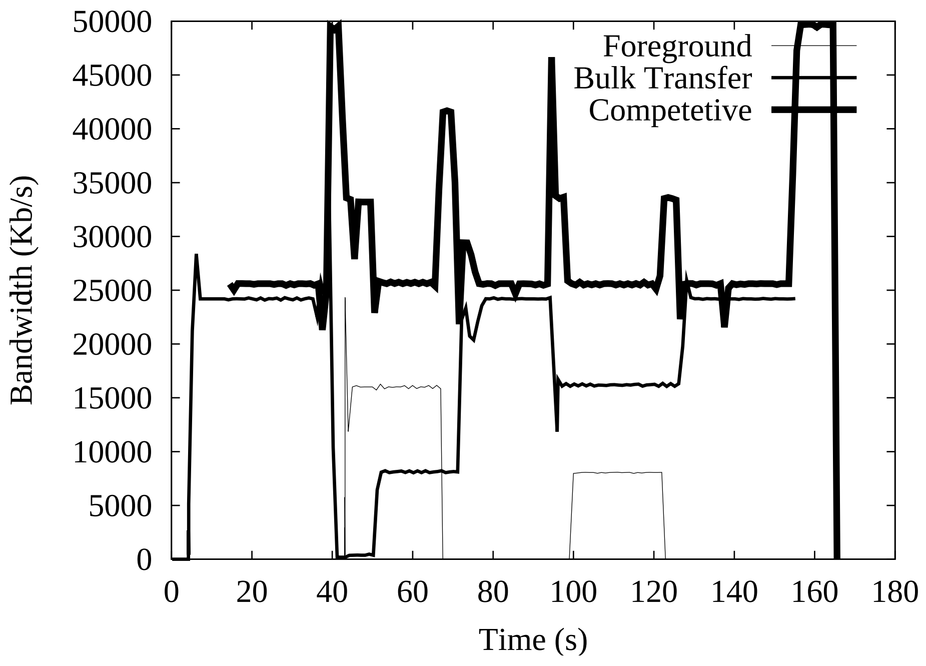

Figure 7.4: Bulk transfer in a wide area network. See text for details.

This gives a proportional share of the bandwidth, and ensures that each flow gets its minimum bandwidth. An optimization could be to decay the minimum bandwidth when the application is able to send at rates greater than the minimum bandwidth, thus allowing more bandwidth to be given to other flows. This allows the links to be shared, while still assisting the bulk transfer flows in finishing as soon as possible.

Although these are important improvements to make, they do not hide our main point: GARA's feedback mechanism can be used in new ways to provide interesting QoS services.

7.2.3 Other feedback services

The previous two sections have described two ways of using GARA's feedback mechanisms. Because GARA has a general-purpose feedback mechanism, it would be easy to provide other useful types of feedback to applications, including notice that a reservation has been preempted. For example, when more complicated policy decision procedures are added to GARA, it may be possible that a reservation for an important person, such as the chair of the department, could preempt reservations for less important people, such as graduate students. In this case, the feedback mechanism could inform applications that their reservation has been revoked.

Another example type of feedback would be notice of over-reservation. If users have to pay for reservations, they would appreciate feedback indicating that their reservation is larger than the capacity they are actually using. A resource manager could detect this situation and use the feedback mechanism to inform the user.

Feedback is useful in building co-reservation agents (discussed below in Section 7.3). If a co-reservation agent makes multiple advance reservations on behalf of a user, it can monitor the reservations before they begin. If a reservation is cancelled because of preemption or because a resource becomes unavailable, the agent can be notified using feedback, and it can react by making finding alternate resources that can be reserved.

7.3 Uniform interface and layering

Claim: The GARA API makes it straightforward to develop higher-level functionality, including reusable, multi-purpose services.

GARA provides an interface that allows users to make single reservations for quality of service. Each reservation can only be for a single resource. Often, this is sufficient because users only need to use a single shared resource which is the limiting factor. For example, a user may have exclusive access to multiple computers and disks, but is required to share access to the network, so only a network reservation is needed.

Some users have more complex demands, such as needing multiple coordinated reservations, or *co-reservations*. Although GARA does not provide co-reservations, it simplifies construction of *co-reservation agents*, because such agents can concentrate on the complexities involved in making multiple reservations without having to also interact with different underlying reservation systems.

We have constructed two different co-reservation systems, to demonstrate the possibilities that GARA provides. Both of these co-reservation systems reside in the high-level services layer described in Section 4.3.1 and interact with the arbitration layer on behalf of the user. Other high-level services can be built on top of these co-reservation systems.

7.3.1 The need for co-reservation

Much QoS research has concentrated on single types of reservations, whether network reservations [15], CPU reservations [35], or disk reservations [37]. However, it is often important to use different reservations at the same time. When multiple reservations are made at the same time, we call it *co-reservation*.

Consider, for example, the scientific visualization application shown in Figure 7.5. Here we have an application reading experimental results from disk, rendering the results by creating lists of polygons, and sending the results to a remote computer which then visualizes the results. If the entire system is manually reserved to be used by the application alone, perhaps by a phone call to a system administrator, then no QoS mechanism is necessary. However, if we are using shared systems, any portion of the system could experience contention, slowing down the scientific visualization. In particular we could have contention for:

- the disk system where the experimental results are stored,
- the CPU doing the rendering,
- the network used for sending the rendered data,
- the CPU displaying the final results.

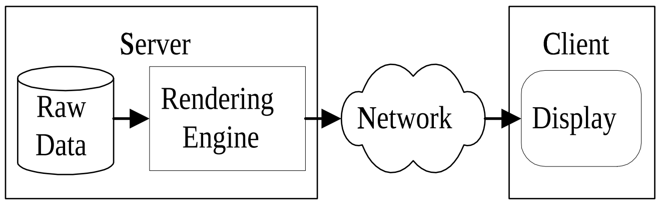

Figure 7.5: An example of an application that could benefit from co-reservation. Because the reservation pipeline uses several different potentially shared resources, it is likely to be beneficial for the application to make a reservation for each resource: disk, graphic pipeline, computer, display, and network.

Any one or a combination of these systems could require the use of QoS. We need to make reservations for each system to ensure that everything works smoothly when we cannot predict what contention will occur in the future.

Figure 7.6 shows a concrete example of the usefulness of co-reservation. In this example, an application is attempting to send data at 80 Mb/s using TCP. Because this is a fast speed, the application can be easily disrupted by contention for the CPU. In the experiment, the application experienced two types of congestion. First, there was network congestion beginning at about time 15 and continuing to the end of the experiment. A network reservation was made at time 40 to correct for this. Second, there was contention for the CPU at about time 60 and continuing for the rest of the experiment. A CPU reservation was made at time 80 to correct for this. From time 80 to 120, both reservations were active, and the application was able to send data at its full rate.

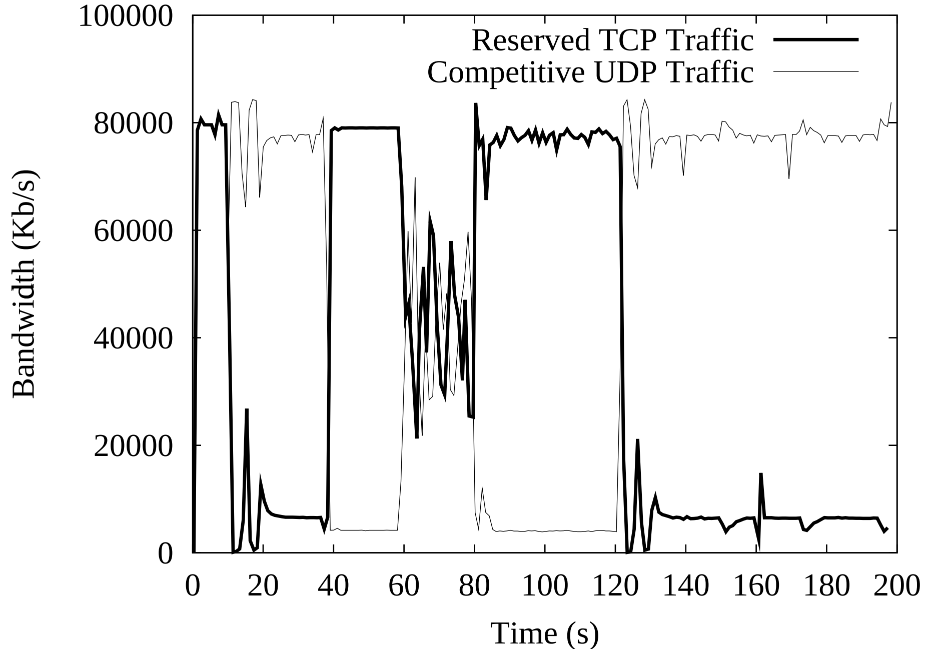

Figure 7.6: Combining DSRT and differentiated services reservations. See text for details.

Although the application shown in Figure 7.6 was an experiment and not performed with a real application, the point that it demonstrates is relevant and important: *It is often* *important to combine different types of reservations in order for an application to achieve its desired performance.*

7.3.2 A generic co-reservation agent

As described in the previous section, there are cases when someone wishes to perform complicated co-reservation tasks. There are at least three factors that can make a co-reservation task complicated:

- *The number of reservations needed.* If more than two or three reservations are needed, it becomes complicated for a user to keep track of all of the reservations. It is easy for the number of reservations to grow large. For example, if a user wants to run a scientific job on two parallel machines, there might be a large number of network reservations between many processes on the two machines, in addition to disk bandwidth reservations for the individual processes.

- *The number of choices.* The user may have a number of choices about which resources to use. For example, a user may need two IRIX computers with at least 256MB of memory, but there may be a large number of IRIX computers to choose from. The choices may become difficult to sort through if, for example, the user also wants to reserve space on a local disk on some of the machines, and not all of the machines have local disk reservations available.

- *Recovering from failure.* While making the reservations, one of the reservation requests may fail, which may require other reservations to be revised. Similarly, if a resource fails sometime after a reservation has been made, the reservations may need to be changed.

We have implemented a co-reservation agent to demonstrate how such a co-reservation agent can work. While this co-reservation agent is relatively simple, it demonstrates the ability to deal with reasonably large numbers of simultaneous reservations and the ability to search through different choices.

Our co-reservation agent is implemented as an API that a programmer can use. It would be easy to wrap this API in a program that users could interact with more directly.

Behavior of the co-reservation agent

The co-reservation agent is program-oriented. That is, it assumes that the user will be running a certain number of programs, and will need reservations that are associated with those programs. Therefore, the user describes his reservation needs as a set of process descriptions, where each process description has a set of reservations that it needs. We suspect this will work well for the majority of co-reservation needs, although some users may not prefer the process-centric view of reservations.

Making a reservation Making a reservation is a five step process:

- The user provides two inputs to the co-reservation agent: the reservations that are needed, and the resources that are available. For instance, the user may request reservations for two computers, and a list of computers that can be chosen from. Actually, the user does not provide the list of available resources directly, but a pointer to where this list can be found. Currently, this list is provided as a file, but it could easily be provided through a directory service, such as the Metacomputing Directory Service (MDS) [17].

- Next, the co-reservation agent refines the possible candidates for compute resources by eliminating the ones that do no have the appropriate characteristics. For instance, if the user requires computers using Solaris 2.8, then computers using Linux can be eliminated.

- Next, the co-reservation agent refines the possible candidates even further by looking up the information that is available about reservations already made on the resources in question. GARA normally reports information about the reservations that have been made—see Section 5.1.4 for more information about reporting.

- Next, the co-reservation agent attempts to make the necessary reservations. Note that, although the co-reservation agent eliminated resources that it knew to be already too busy to accommodate the desired reservations, the resources may still refuse the reservations. This could be for a number of reasons: other reservations may have been made just after the co-reservation agent retrieved information about the reservations or there could be a policy that declines the reservation for a reason that cannot be known by the co-reservation agent. If a reservation request does not succeed, the co-reservation agent uses depth-first search to choose other resources.

- Finally, if all of the reservations succeed, the co-reservation agent creates a *unified reservation handle*. This acts just like the reservation handle described in Section 4.3.2, but it encompasses enough information to recreate all of the reservation handles associated with each of the reservations that were made. Just like individual reservation handles, the unified reservation handle can be saved to disk, transferred from program to program, and used in future API calls.

Describing reservation needs One of the inputs that the user provides to the co-reservation agent is the list of reservations needed. An example of this input is in Figure 7.7. Note that these needs match the example in Figure 7.5.

```
Compute ‘‘server’’
  Architecture Sun
  OS Solaris2.8
  Network ‘‘client-resv’’ tcp to client 20Mbs
  Disk ‘‘server-disk-resv’’ 20MB
Compute ‘‘client’’
  Architecture Sun
  OS Solaris2.8
  Percent-CPU 0.7
  Network ‘‘server-resv’’ to server 100Kbs
```

Figure 7.7: Co-reservation needs as provided to the co-reservation agent

There are two important things to notice about this list of needs. First, the needs are organized around a list of the processes that will be used. Second, each of the needs and processes have a unique name (the text in quotes) by which the user can refer to them later.

Launching programs When the user is ready to launch his processes, the co-reservation agent handles the launching. The co-reservation agent uses the GRAM library [11], available as part of the Globus Toolkit [20]. The user merely provides a standard Globus Resource Specification Language (RSL) string describing each of the processes, but associates each one with the name of the need that was provided when the reservations were described. The co-reservation agent then simply launches each process.

Binding the reservations As described in Section 4.3.2, reservations need to be claimed before they can be used. The processes that are launched can use the same co-reservation API to bind the individual reservations by providing the name of the reservation and the parameters that are needed to claim the reservation. The co-reservation API then properly binds the reservation.

Extensions The co-reservation agent is useful as it is, but there are several ways in which it could be improved. In particular, the co-reservation agent could support higher-level descriptions of the needs of applications, such as "I want to send MPEG-2 video encoded with the Sorenson codec at 20 frames per second", and it could translate that into the necessary bandwidth reservation. In general, this turns out to be a rather complicated task worthy of extra research, particularly given the wide variety of applications that are not multimedia applications—how does one characterize the bandwidth needs of a distributed MPI-based climate modeling program?

Another enhancement to the co-reservation is resource discovery. The agent could discover resources in a more flexible manner than it currently does, perhaps by querying directory services as well as directly querying potential resources.

Finally, the co-reservation agent could be an independent process that users connect to when they need operations with co-reservations to be performed. The advantage to this is that when advance reservations are needed, the co-reservation agent can reliably track the reservations and respond to failures and the client process and machines do not have to stay active.

7.3.3 Multi-domain network reservations

As noted in Section 6.5.1, our differentiated services network resource manager only controls the network in a single domain. While this is certainly useful, it is desirable to have network QoS for traffic passing through multiple domains. In this section, we will consider one method for handling multiple domains.

Consider the situation in Figure 7.8. Here we have a server in Network 1 sending traffic to a client in Network 3. The server and client are in separate networks, and there is an extra network between them as well, perhaps the network of a Internet Service Provider (ISP).

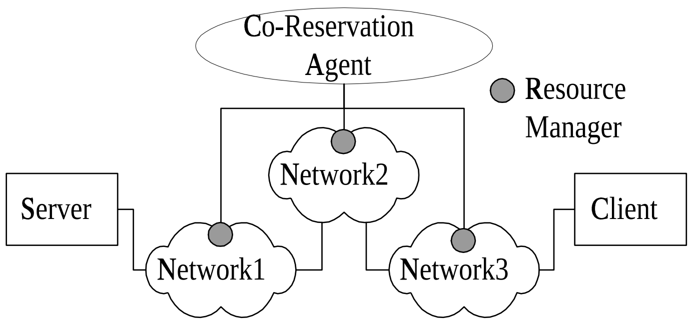

Figure 7.8: Simple network co-reservation using a co-reservation agent. In this example, the co-reservation agent is making a reservation for the network connection between the server and the client.

Each of the three domains belongs to a different administrative entity, and it is unlikely that any of these administrative domains wishes any other administrative domain to have the ability to perform admission control for them or to configure their routers. Therefore, each domain has its own differentiated services resource manager, and a reservation has to be made with each resource manager in order to create an end-to-end network reservation from the server to the client.

This situation is similar to the one that the co-reservation agent described in Section 7.3.2 solves. Just as that co-reservation agent makes a reservation for each resource that is needed, all we need to do is make a reservation in each of the three domains. Because the co-reservation agent described in Section 7.3.2 is process oriented, it cannot be directly used for the multi-domain network co-reservation, but the principle is exactly the same, and is shown in Figure 7.8. The agent simply contacts each network's resource manager and requests a reservation through the network. These reservations can be made in parallel to decrease the total time it takes to make the end-to-end reservation. If all of the reservations are accepted, then the agent can return a unified reservation handle for the end-to-end network reservation, otherwise it reports failure.

We have implemented such a network co-reservation agent. It works similarly to the process oriented co-reservation agent described in Section 7.3.2.

Unfortunately, there are disadvantages to this simple method. First, there is no way to verify that a user requests a reservation in each domain encountered along the end-to-end path, and this might cause other users' reservations to not receive the bandwidth that was guaranteed to them. Skipping domains in this way could be caused by either a broken or malicious co-reservation agent. To understand why skipping domains could cause this problem, look at Figure 7.9.

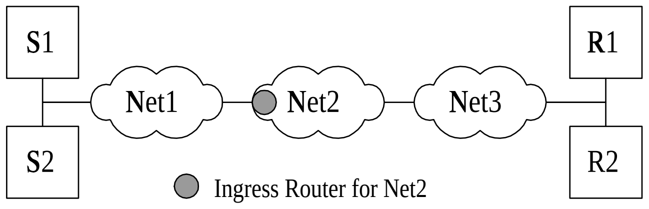

Figure 7.9: This is a scenario that can cause difficulty with the simple network co-reservation. See text for details.

Assume that, in this figure, S1 is sending data to R1 and S2 is sending data to R2. Both of them are sending at 10 Mb/s. A reservation has been made for (S1,R1) in each of the three network domains, but the reservation for (S2,R2) has only been made in networks 1 and 3, not in 2.

What happens in this case? The traffic from S1 and S2 will traverse network 1 with no difficulty, but network 2 will not be expecting 20 Mb/s of reserved traffic, so the ingress router in network 2 will police the traffic at a rate of 10 Mb/s instead of 20 Mb/s. (Such policing is not strictly required, but is common in differentiated services networks.) That means that some portion of both traffic streams will be dropped, since the ingress router is policing based on the differentiated services code point, not on the individual reservations. Therefore a portion of the traffic from S1 to R1 will be improperly dropped, and the guarantee for the reservation will not be fulfilled, all because the reservation for (S2, R2) was incorrectly made.

Another disadvantage to this method of co-reservation is that it does not scale well beyond medium-sized grid environments because it requires the co-reservation agent to have a knowledge of the domain to domain routing, which becomes unmanageable when a large number of networks are involved. Although this is a disadvantage, it is not currently as large a disadvantage as one might think because for the time being, rather few networks support differentiated services, so there are no large scale networks that need to be supported. Also, the design of more scalable solutions to end-to-end network reservation is not well-understood (although we sketch a solution below). Therefore this is a good, interim scale solution that allows significant experimentation while differentiated services network become more widely deployed and we begin to understand larger scale solutions.

In collaboration with other researchers, we have been involved in developing a scalable end-to-end network co-reservation mechanism that solves these problems [8, 49]. Although we have not described such solutions here, it should be clear that GARA enables a variety of methods for providing co-reservations, because it provides a flexible and uniform substrate for building services that insulates these services from the difficulties of interacting with QoS systems.

7.4 High-level programming

Claim: GARA mechanisms are sufficiently abstract that they can be incorporated easily into higher-level programming models.

It should not be surprising that we, the developers of GARA, found it rather easy to use GARA within programs to gain access to QoS. It is much more interesting to ask if GARA can be used easily by other programmers. Towards this end, we worked with the developers of an implementation of the Message Passing Interface (MPI). This had two results. First, we were able to discover that other programmers found GARA easy to work with. Second, we were able to make network QoS available to high-performance scientific applications, without the programmers of these applications having to understand GARA at all.

In our experience, scientific programmers prefer to not work with the common sockets-based APIs, but prefer higher-level communication libraries such as MPI. Instead of just the simple read/write primitives provided by socket-based APIs, MPI provides communication primitives that simplify parallel programming such as broadcast (send to all nodes) and gather (receive from all nodes) operations. Adding support for QoS to MPI allows programmers to work within their familiar environment.

Our prototype MPICH-GQ implementation combines elements of the MPICH-G2 (formerly MPICH-G) wide area implementation of MPI [27, 19] with GARA. Experimental studies with simple MPI benchmark studies demonstrate our ability to deliver high performance in the face of network contention.

Note: This section is a revised version of a paper [48] that was nominated for both the Best Paper and Best Student Paper Awards at the Supercomputing 2000 Conference.

7.4.1 Quality of service and MPI

The communication structures associated with MPI applications are often significantly more complex than in media applications (such as audio and video), in three principal respects:

- Communication is often bursty: an application may compute for a while, then call a communication function, then compute some more. In some cases, communication can be overlapped with computation, but in others, computation ceases until communication completes. Furthermore, communication structures and rates may not be predictable.
- Communication is typically achieved via reliable protocols such as TCP. These protocols further complicate the communication structure, because a single application-level message may result in many low-level communications, and packet loss may trigger unexpected behaviors.
- Communication can involve many processes, rather than a single pair.

To illustrate the implications of these differences, consider a simple finite difference application partitioned across two 8-processor multiprocessors connected by a wide area network. A simple calculation of the total data volume exchanged by the application suggests that the application maintains an average data rate of 1 Mb/s. Yet if we configure our network to support a premium flow at this rate, we find that things do not perform as we expect. The application immediately performs an *MPI_Send* involving a large buffer (100 KB), depleting the token bucket and causing packets to be dropped. TCP enters into slow start mode and starts sending more slowly, gradually building up its send rate until packets are dropped again. The result is an extremely low communication rate and an underutilized network. The provision of QoS for such applications requires new methods and mechanisms.

Figure 7.10 illustrates the types of problems that can arise. Here, we deal with a simple TCP program that is attempting to send data at approximately 50 Mb/s over a congested network, with a reservation that is somewhat too low (40 Mb/s). As we see, the bandwidth obtained by this program varies wildly: every time TCP kicks into slow start mode, the bandwidth drops significantly, then slowly increases until packets are dropped again.

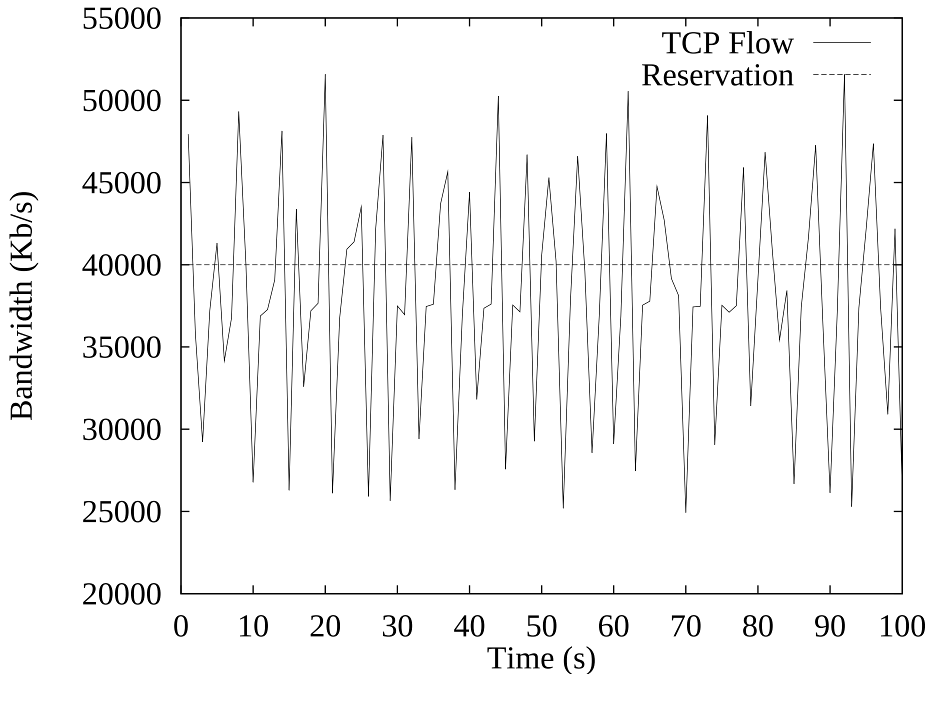

Figure 7.10: An application using TCP has made a reservation for only 40 Mb/s, when it is sending at 50 Mb/s.

The fact that a typical MPI program may involve large numbers of communicating processors complicates things further. We need to bind all relevant flows with underlying QoS mechanisms; in addition, multiple concurrent TCP flows can lead to some interesting inter-flow interactions.

7.4.2 MPICH-GQ

MPICH-GQ extends the MPICH-G2 wide area implementation of MPI and leverages mechanisms provided by the GARA QoS architecture to deliver QoS support for MPI applications.

Figure 7.11 shows the principal components of the MPICH-GQ architecture. These are as follows:

- The *MPICH implementation of MPI* [27] is extended in a standards-compliant fashion so that MPI's attribute mechanisms can be used to communicate with the underlying QoS system. (Note that we have prototyped this, but the results presented used a slightly different mechanism.)
- An *MPI QoS Agent* incorporates the rules used to translate application-level QoS specifications into the lower-level commands and parameters required to implement QoS.
- As in MPICH-G2, a *Globus device* [19] provides low-level security, startup, and other functions for wide area networks.
- The *Globus I/O library* provides a convenient wrapper for the low-level socket calls used to implement wide area transport; traffic shaping can also be performed here.
- The *GARA system* [22] is used to reserve premium bandwidth and to control physical devices such as routers and computers.
- The physical devices themselves are controlled via their implementations of *differentiated services* and other mechanisms.

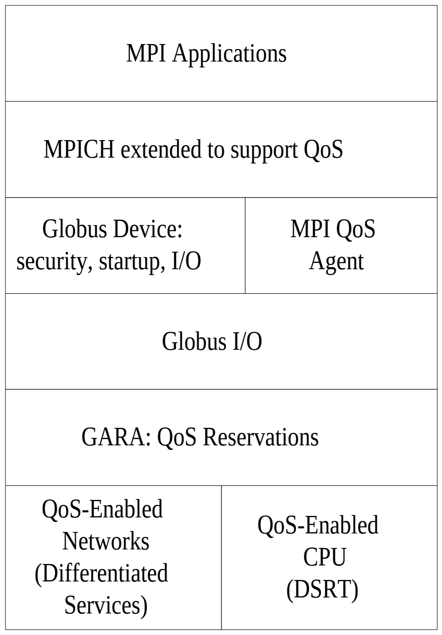
MPI Applications
MPICH extended to support QoS
Globus Device: security, startup, I/O
MPI QoS Agent
Globus I/O
GARA: QoS Reservations
QoS-Enabled Networks (Differentiated Services)
QoS-Enabled CPU (DSRT)

Figure 7.11: The MPICH-GQ Architecture

At the time of writing, we have prototyped significant fractions of the MPICH-GQ architecture—enough to conduct the experiments described below—but do not have a complete implementation. The major component that we have not yet constructed is the MPI QoS Agent. As we describe in the next section, we currently bind QoS parameters directly to application-level flows.

Application-level QoS specification

The Message Passing Interface (MPI) standard was designed to support high-performance, scalable message passing for communication between two or more processes. The major parts of this programming model are well known (see [28, 26, 29]).

A goal in designing MPICH-GQ was to make QoS capabilities available within this standards-based framework. One consequence of this goal is that we cannot extend MPI arbitrarily. For example, it might be convenient to introduce an *MPI_Set_qos* function, but MPI programs that used it would no longer be standards compliant or portable to different MPI implementations.

Fortunately, the MPI standard provides an elegant solution to the problem of enabling application-level tuning without compromising portability, namely its *attribute* mechanism. This part of the MPI specification was introduced with the specific goal of allowing users and implementers to share information to enable faster or more reliable communication.

In the MPI programming model, all communication takes place within a *communicator*. A communicator is simply a group of processes, with an additional, unique communication context that ensures that messages sent in one communicator cannot be received in another communicator.

The application programmer can create, set, or get *attributes* that are maintained on a communicator-by-communicator basis. An attribute is identified by an integer *keyval*. The value of an attribute (in C and C++) is a pointer, thus providing a standard-conforming way of retrieving information from the MPI implementation (`MPI_Attr_get` with a predefined `keyval`) and providing information to the MPI implementation (`MPI_Attr_put`).

MPICH-GQ exploits this attribute mechanism to exchange information between the user's application and the MPI implementation, using `MPI_Attr_put` to specify required QoS and `MPI_Attr_get` to see whether the requested QoS is available. Because attributes are specific to a particular communicator, it is possible, by careful creation of appropriate communicators, to target both queries and requests to specific links or sets of links. Note that the action of putting the attribute actually triggers the request for QoS, which is slightly different than the normal usage of attributes, which do not trigger actions.

In our work with MPICH-GQ, we focus initially on QoS attributes that are applied to two-party intercommunicators and on the techniques required to communicate low-level specifications of required QoS to the underlying QoS system. A typical specification is illustrated in Figure 7.12. The QoS class may be "best-effort" (i.e., no QoS), "low latency" (suitable for small message traffic [51, 50]: e.g., certain collective operations), or "premium." The maximum message size allows us to translate application reservation sizes to network reservation sizes, because it is possible to calculate the amount of protocol overhead. Extensions to ensembles of processes will be considered in the future, as will more interesting mappings from QoS specifications expressed in terms meaningful to MPI programmers, such as MB/s or messages per second.

```
struct qos_attributes
{
  u_int32_t qosclass;
  /* Peak bandwidth in kbps */
  double    bandwidth;
  /* Max size used in MPI_Send */
  int       max_message_size;
  char      *gatekeeper;
} attributes;
    ...
MPI_Attr_put( comm, MPICH_QOS, &attributes );
MPI_Attr_get( comm, MPICH_QOS,
              &attributes, &flag );
```

Figure 7.12: QoS-enhanced MPI code to set and check the QoS parameters associated with a communicator.

The features just described allow for QoS specification internal to an MPI application. In addition, it can be useful to allow for external management of QoS by a separate QoS agent. To support this feature, we also define a function that can extract the necessary information (basically port and machine names) from a communicator.

Support for TCP flows

An MPICH-GQ call to GARA that requests the reservation of network resources for an MPI application flow must ultimately be translated into calls to resource-specific control functions to configure the routers (and/or CPU schedulers, etc.) that implement QoS functions. This configuration process is complicated by the fact that the application-level traffic consists of one or more high-performance TCP flows. TCP's flow control and congestion control mechanisms [53, 10], while critical to the effectiveness of TCP in shared networks, have the unfortunate consequences of making TCP traffic both bursty and sensitive to the loss of individual packets [38, 34]. In a differentiated services-based system, this means that we need both a large token bucket on the edge router and an accurate reservation value.

The GARA differentiated services resource manager (see Section 6.5.1) incorporates configuration rules that allow it to set these values correctly. In brief, we configure the token bucket depth to be

$$depth = bandwidth * delay,$$

where “depth” is in bits, bandwidth is in bits per second, and “delay” is in seconds. However, the token bucket is usually specified in bytes, not bits, so the formula becomes:

$$depth = bandwidth * delay * 8.$$

In our local testbed, the delay is small: on the order of a millisecond or two. A two millisecond delay would therefore suggest that the depth of the bucket should be

$$bandwidth * \frac{2}{1000} * 8 = bandwidth/62.5.$$

However, to allow for larger bursts in traffic, we currently use $bandwidth/40$. As we explain in Section 7.4.3 below, this value is not always adequate.

7.4.3 Experimental results

We present experimental results that demonstrate our ability to deliver QoS to MPI applications and also expose some of the difficulties that one encounters when dealing with bursty MPI traffic.

Our experimental configuration is the same testbed that was described in Section 6.5.1, and further description can be found there.

QoS and MPI: ping-pong

We first present MPICH-GQ results for a simple “ping-pong” program, in which two processes repeatedly exchange a fixed-sized message via `MPI_Send` and `MPI_Recv` calls. While artificial, this communication pattern is characteristic of many SPMD applications.

Figure 7.13 shows the one-way throughput obtained by this program as a function of reservation size, for four different message sizes, in the face of heavy contention. Contention is generated via a UDP traffic generator that is capable of overwhelming any TCP application that does not have a reservation. (As the two processes exchange messages, total "throughput"—and reservation—is twice what is shown here, when summed over both directions.) We do not show the results obtained in the absence of a reservation or in the absence of contention (and with no reservation), but, in brief, performance is extremely poor in the first case but is at the peak levels reached in the figure in the second case.

We see that the achieved throughput improves as the applied reservation increases until the reservation is "adequate" for the message size in question, after which further increases in reservation size have no significant impact. This is the general behavior that we would expect: when the reservation is too low, packets are dropped. In fact, the throughput that was observed was much lower than the reservation, until the reservation was large enough. This is because TCP backs off when packets are dropped, as discussed above.

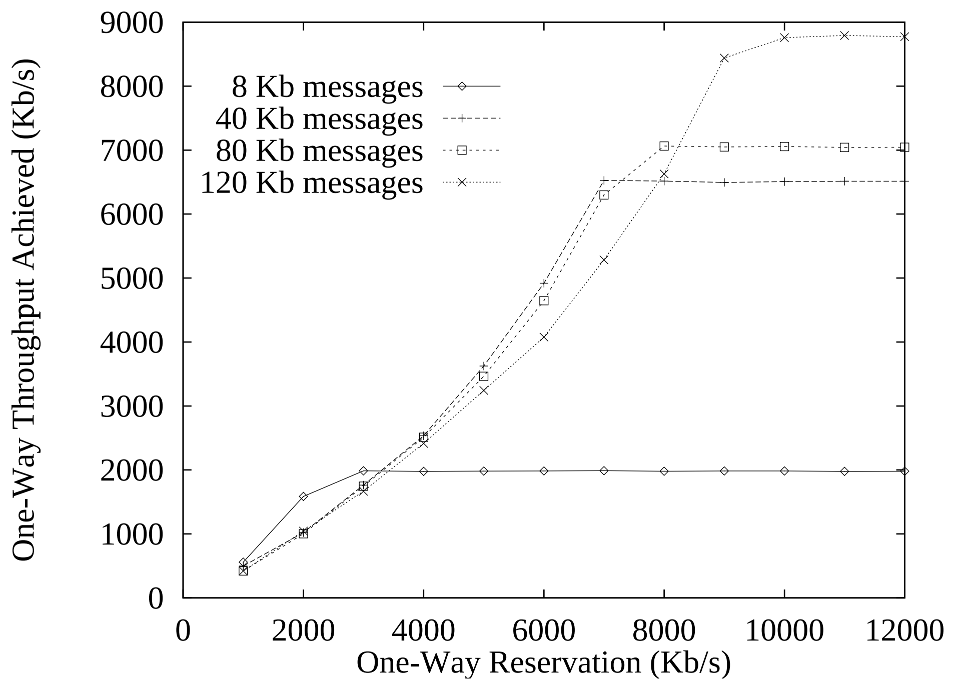

Figure 7.13: The effect of different reservation sizes for the ping-pong MPICH-GQ program. Each line represents the throughput achieved for a particular message size at different reservation sizes.

QoS and MPI: distance visualization

Our next results are for an MPI program designed to emulate a distance visualization pipeline. The program communicates a stream of fixed-sized messages from a sender to a receiver at a fixed rate; both the rate (“frames per second” or fps) and the message size (“frame size”) can be adjusted, hence varying both the generated bandwidth and the burstiness of the traffic.

Figure 7.14 shows the throughput achieved by this program as a function of reservation size for frame sizes of 40, 80, 160, and 240 Kb. (The rate was fixed at 10 frames per second.) Once again, we see that the achieved throughput increases with reservation until the reservation is “adequate.” However, in contrast to the ping-pong case, we see that the performance at lower reservations is significantly worse than we would expect from simple scaling. This effect is due to TCP congestion control strategies. We also see that we require a reservation value of around 1.06 of the sending rate, because of TCP packet overheads.

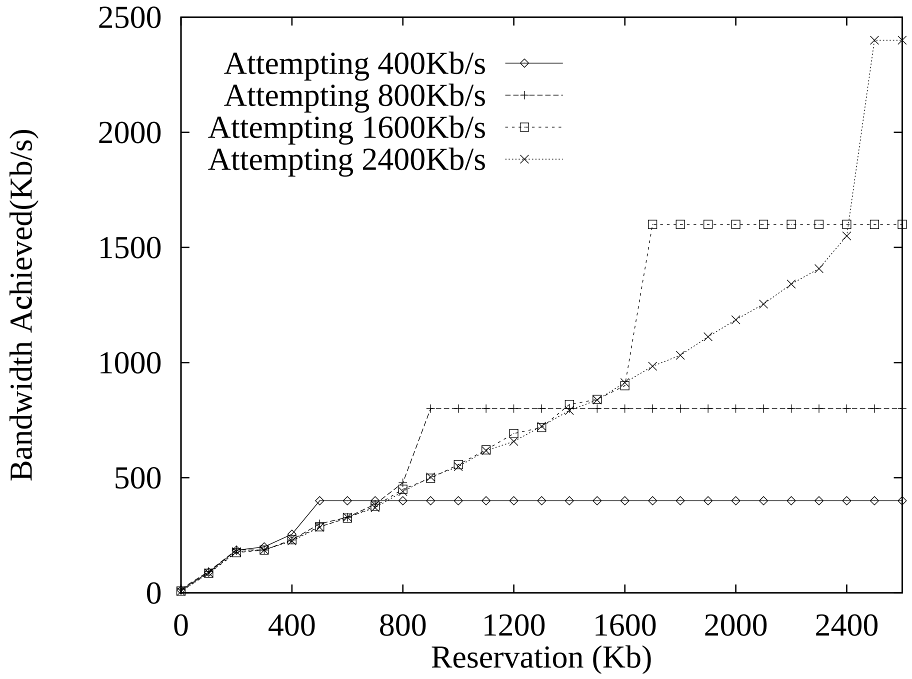

Figure 7.14: The effect of different reservations on the visualization application attempting different throughputs. Note that making a reservation that is even a little bit too small dramatically decreases the throughput that is achieved.

The effect of burstiness

We outlined in Section 7.4.2 how MPICH-GQ currently attempts to deal with small bursts of TCP by adopting a moderately large, but fixed, value for the size of the token bucket. We present results here that demonstrate the impact that this value can have on performance.

In the experiments described, we used our visualization program to transmit data at various rates, while varying both the burstiness of the traffic (1 frame per second or 10, with the former featuring bursts that are ten times as large) and the size of the token bucket ($bandwidth/40$: “normal” and $bandwidth/4$: “large”).

The results, shown in Table 7.1, demonstrate that there are limits to the size of the burst that our “normal” token bucket depth can deal with: with the normal depth, the bursty configurations needs an approximately 50% larger reservation.

Figure 7.15 provides an aid to visualizing the difference in burstiness between the two programs. Note how the program running at ten frames per second has much smaller bursts that are well spread out, while the program running at one frame per second sends all of its data in one much larger burst, thus effectively giving it a larger bandwidth over a small time interval.

These results present serious challenges for MPICH-GQ design. One approach to this problem is to attempt to compute the “correct” token bucket size dynamically, by using application-specific information and perhaps also dynamic network performance data [57]. However, collecting such dynamic data also expends scarce system resources. An alternative approach is to incorporate traffic-shaping support into the MPICH-GQ implementation on the end-system.

Combining network and CPU reservations

Up to this point, we have only considered network QoS for MPI programs. Unfortunately, as discussed in earlier chapters, it is not always sufficient to rely on network QoS. For example, if there is contention for a CPU or disk, it may be necessary to use QoS mechanisms to control access to the CPU and disk to ensure end-to-end QoS.

We have done experiments to demonstrate this necessity. In order to create and enforce CPU reservations we are using the Dynamic Soft Real-Time CPU Scheduler [9]. DSRT

Table 7.1: The reservation required to achieve a specified throughput, for varying degrees of "burstiness" (expressed in frames per second) and token bucket sizes. All bandwidths and reservations are in Kb/s.

	Reservation Required		
Bandwidth	Normal Token Bucket		Large Token Bucket
Desired	10 fps	1 fps	1 fps
400	500	750	500
800	900	1450	900
1600	1700	2700	1700
2400	2500	3600	2500

works by overriding the Unix scheduler and performing soft real-time scheduling of select processes. (See Section 6.5.3 for more information about DSRT.)

Figure 7.16 again shows a trace of our visualization application. At the beginning, it is able to maintain a fairly steady throughput of 15Mb/s. However at 10 seconds, a CPU-intensive application begins running on the same machine as the sending side of the visualization application. This reduces the bandwidth significantly, so a CPU reservation for 90% of the CPU is made at 20 seconds, and the visualization application again is able to achieve its full bandwidth.

There are some interesting aspects to this example. When we first developed our visualization application, our implementation of MPI was using TCP socket buffer sizes of 8KB but was writing to the socket in chunks greater than 60KB. This had the effect of using a large amount of user CPU time (as opposed to kernel time), so the effect of the CPU congestion was more pronounced at smaller bandwidths. When we began using larger socket buffer sizes, we had to significantly increase the bandwidth that the application was using before the bandwidth was affected by CPU congestion. This was because the network communication was actually kernel time, not user time. In addition, our visualization application was originally an inaccurate simulation of a visualization application: it sent a chunk of data, slept for a short time, then repeated. Since the network writes were blocking, the application actually used little CPU time, and was not significantly affected by the CPU contention. After a modification to make the visualization application do some "work" between sending frames, it was more affected by the CPU contention.

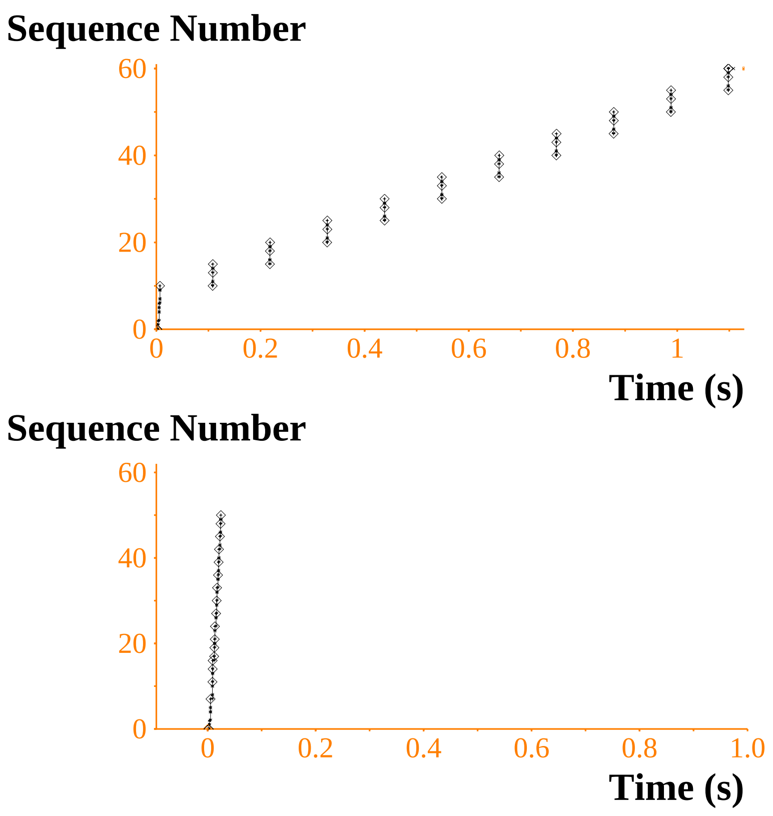

Figure 7.15: TCP traces of two programs that each send at 400Kb/s, but with different burstiness characteristics. On the top is a program sending 10 frames per second, and each frame is 40Kb. On the bottom is a program sending just 1 frame per second, and the frame is 400Kb. (This corresponds to the first line of Table 7.1.) In each case, only one second of the program's execution is shown.

There are two lessons to draw from this experience. First, applications that use TCP and want high performance need careful tuning (such as socket buffer sizes) to actually obtain the high performance. Since MPICH-GQ applications do not use TCP directly, that burden falls on MPICH-GQ directly. Second, it can be difficult to decide how best to optimize a program: does it simply need to have TCP parameters tuned (a network optimization), or does it need a CPU reservation (a CPU optimization), or does it need both? Applications that have large bandwidths are much more sensitive to CPU contention, and may need CPU reservations to achieve their desired performance.

Figure 7.17 shows another example of CPU reservations. In this case, the application which is trying to send data at 35Mb/s encounters both network congestion and CPU

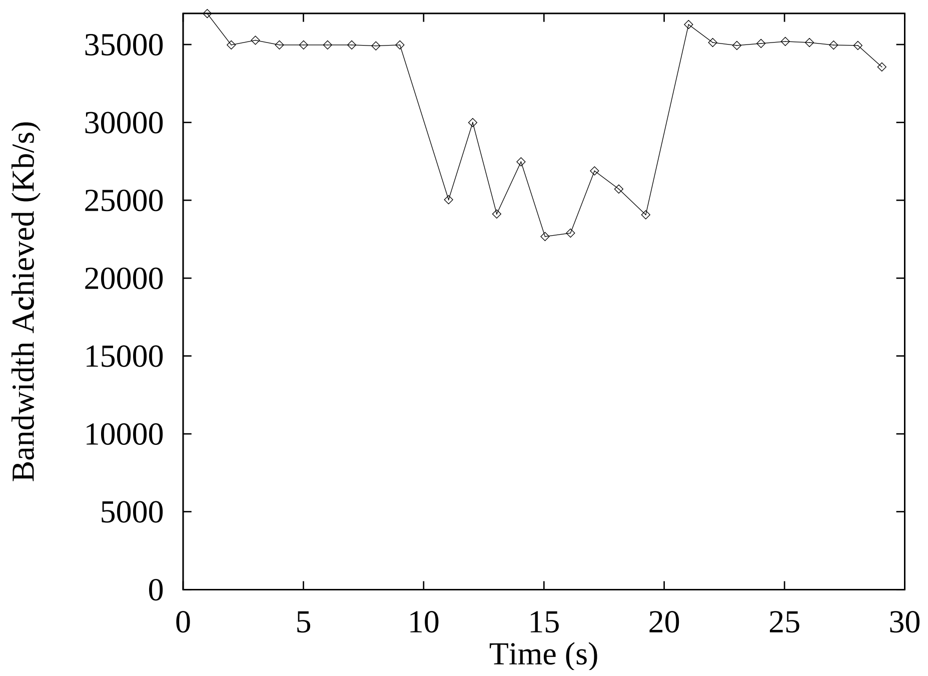

Figure 7.16: The bandwidth achieved by the visualization application. Contention for the CPU on the sending side begins at 10 seconds, and a reservation is made at 20 seconds.

contention. The network congestion begins at time 10 and continues to the end of the experiment, while the CPU congestion begins at time 30, and continues to the end of the experiment. Both network and CPU reservations are made to overcome the resource contention. This figure demonstrates that not only can network congestion and CPU contention combine to decrease an application's bandwidth, but it is possible to overcome such contention in order to achieve good performance. Note that it is insufficient to make just a network reservation or a CPU reservation: both reservations are needed.

Applications that use MPI often assume that they have exclusive access to a machine. If exclusive access can be ensured with non-QoS mechanisms, then there is no need for using systems like DSRT. However, it is clear that there are times when combining network and CPU QoS mechanisms is advantageous.

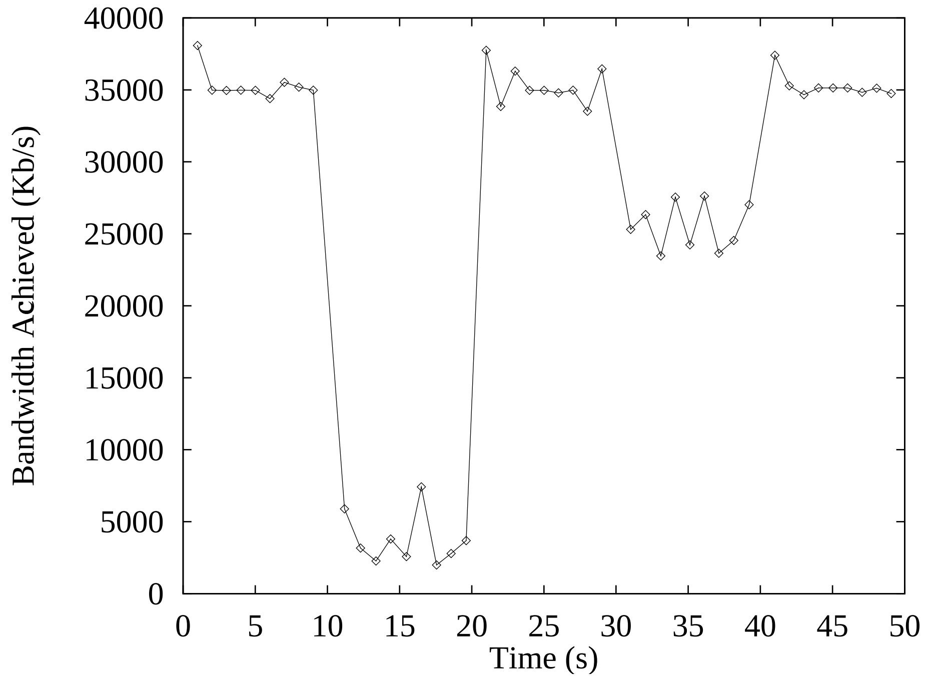

Figure 7.17: A trace of the bandwidth achieved by the visualization application as it attempts to achieve a constant 35Mb/s rate. Initially it runs well (0-10 seconds), then network congestion affects its bandwidth (11-20 seconds) until a network reservation is made (21-30 seconds). Bandwidth again decreases when there is CPU contention at the sender (31-40 seconds) until there is a CPU reservation (41-50 seconds).

Demonstration with a real application

Although all of the experiments described above were done with simple testing tools, we have used MPICH-GQ in a real scientific visualization application, similar to the artificial visualization program described in Section 7.4.3. We demonstrated this application at the IEEE/ACM Supercomputing 2000 Conference in Dallas, TX.

The demonstration was unique but straightforward. We used a scientific visualization program that would send visualizations from server computers to visualization computers across a network. A single movie can be displayed across on one to six different monitors. Each additional monitor requires the addition of an extra server/visualization computer pair, and increases the resolution of the movie being viewed.

For our particular demonstration, we showed two simultaneous movies, each showing on two monitors, as shown in Figure 7.18. Each movie was uncompressed and ran at about 50Mb/s. When network congestion was added, each movie clearly slowed down to a crawl. When the QoS was turned on for one of the movies, that movie ran at full speed while the other movie continued to move at a crawl. While we have no quantitative results to show from this demonstration, the qualitative effect was clear and obvious: QoS really can be provided to high-performance applications. Equally as important, it was trivial for the application developers, who are visualization gurus, not network gurus, to add support for QoS by using MPICH-GQ.

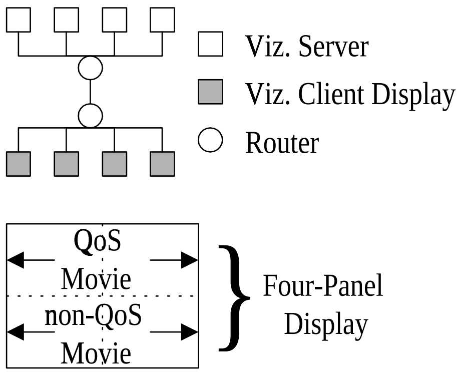

Figure 7.18: The setup for the MPICH-GQ demonstartion. See text for details.

CHAPTER 8
FUTURE WORK AND CONCLUSIONS

8.1 Future Work

GARA provides many opportunities for future work. Because GARA is a modular system, it is easy to add additional QoS mechanisms to support applications or to experiment with alternate QoS mechanisms. Because GARA unifies different types of QoS under a single umbrella, it is easy to build interesting new services that use GARA. Most importantly, because GARA already provides a useful range of services, it is ready to be deployed for regular use, and such deployment would provide valuable lessons. We will now look at several interesting options for future work that GARA enables.

8.1.1 New types of QoS

Because GARA is a flexible, modular system, it practically begs for expansion. There are a number of expansions that would be interesting to make, including:

- *Different Network QoS* There are different underlying QoS mechanisms that can be experimented with using GARA. For instance, we could experiment with using alternate types of differentiated services, such as Assured Forwarding [30]. We could experiment with Multiprotocol Label Switching [47], which is a promising technique for not only speeding up packet routing, but can be used for providing QoS. GARA makes it easy to develop these alternate network QoS mechanisms as well as other ideas that develop in the future.

- *Different CPU QoS* GARA currently works with DSRT for real-time CPU scheduling, but it would be worthwhile integrating with other systems like RT-Mach [39]. Similarly, GARA works with PBS for advance reservations on batch systems, but it could also work with the Maui batch scheduler.

- *Different types of QoS* GARA could provide different types of QoS, such as guaranteed access to databases, exclusive access to software licenses, and other types of QoS that will become apparent in the future.

8.1.2 Enhanced co-reservation

Currently the co-reservation agents built using GARA (see Section 7.3) are useful, but relatively simple. Because GARA makes it easy to use different types of QoS, it enables the development of such co-reservation agents. Agents could be built to optimize aspects of the reservation, such total delay, cost, etc.

8.1.3 Policy

A common question that is asked about GARA is something like, “Well, what if there is no room for an additional reservation, but the president of the university wants a reservation?” The question comes in many forms, but at its core it points out an important area for future work: the need to develop more flexible policy.

Currently, GARA implements only the simplest of policies: if a user can be authenticated and is listed in the “allowed” list of users, then a reservation can be made. Clearly, more interesting policies could be enabled. Are some users only allowed to make reservations on weekends? Can some users preempt other users? Can reservations have a minimum or maximum request? Development of such interesting policies would be an excellent enhancement for GARA. A good candidate for a policy mechanism would be a role-based policy system [14].

8.1.4 Use in the real world

Although all of this future work is interesting and worthwhile, there is other work that would do more to advance GARA in useful ways: using GARA in the “real world”. For example, if we integrated GARA with a regularly used video-conferencing system that required CPU and network QoS, we could gain invaluable experience in using advance reservations and QoS on a daily basis. This would help in several important ways:

- *User Interface* How do users want to interact with advance reservations of multiple resources at the same time? It is unlikely that users will ever want to see the reservation handles described in Chapter 4, or specify times in seconds since 1970. It is likely that significant effort needs to be put into building an infrastructure to help people interact with QoS and advance reservations. For example, we may want a secure database for tracking reservation handles, so users can easily find and use the reservations they have made.

- *System Stability* System stability is more than simply fixing the bugs that inevitably occur in the software. Does GARA scale well when tens or hundreds of users are actively using it? How well do our low-level network QoS mechanisms work when we use a production network that is heavily and unpredictably used? While we feel that GARA is well-designed, there is no test like real-world usage to learn where the faults lie.

- *Acceptance* As interesting as GARA is, not very many people use it, and the users are all researchers. If people are exposed to the benefits of QoS systems, it will push the development of better QoS systems, because people will actually want to use them. This is a critical factor in the development of QoS systems.

8.2 Conclusions

As the previous chapters have shown, I have made three important contributions in this dissertation:

The main contribution of this dissertation is an innovative, modular, and extensible QoS system architecture (GARA) that ties together different QoS mechanisms. This architecture is not merely a simple blending of mechanisms, but an interesting contribution in its own right.

This dissertation also describes two other contributions. First, we have provided significant examples of how to simplify access to QoS: We have extended an implementation of the widely-used Message Passing Interface (MPI) to allow programmers to easily request network QoS, and we have demonstrated methods of combining multiple reservations. This sort of integration is critical in making QoS more widely accessible. Second, we have added to the understanding of mechanisms that can be used to provide QoS, particularly for network QoS using differentiated services.

The combination of these three contributions demonstrates that it is possible to provide a unified QoS system that is convenient for programmers to use and provides a useful capability to high-end applications.

APPENDIX A
THE GARA API

A.1 Using GARA

Client programs access GARA through a library written in C. Any language that can link to C libraries can use GARA. There is also a Java implementation, but it is not described here. To use GARA, you will first need to have linked your program with these libraries. You will need to include "globus_gara_client.h" to provide prototypes for the GARA functions and constants. You will also need to include "globus_common.h", to gain access to `globus_module_activate()` and `globus_module_deactivate()`.

A.1.1 Initializing GARA

Before you can use GARA, you need to initialize it. GARA is initialized like other modules in Globus, using `globus_module_activate()`:

```
globus_module_activate(GLOBUS_GARA_CLIENT_MODULE);
```

A.1.2 Describing a reservation request

Reservation attributes are described using the Resource Specification Language (RSL). An RSL string is simply a list of attribute-value pairs that looks like:

```
&(attribute-1=value-1) (attribute-2=value-2) ...
 (attribute-N=value-N)
```

An example RSL string for requesting a network reservation for 150 Kb/s between looks like this:

```
&(reservation-type=network)
  (start-time=953158862)
  (duration=3600)
  (endpoint-a=140.221.48.146)
  (endpoint-b=140.221.48.106)
  (bandwidth=150)
```

Note that this string was spaced out on several lines for readability, while RSL strings do not have newlines in them.

Table A.1 lists attributes that may be used to specify a reservation. The universal attributes are for all types of reservations, while the other attributes are for specific types of resources. Note that the compute resource attributes are mutually exclusive.

A.1.3 Creating a reservation

Before you can create a reservation, you will need to describe your reservation, as described in Section A.1.2 above. Then you request your reservation with `globus_gara_reservation_create()`:

```
    int  error;
    char *request_rsl = "&(reservation-type=network)"
                        "(start-time=953158862) (duration=3600)"
                        "(endpoint-a=140.221.48.146)"
                        "(endpoint-b=140.221.48.106)"
                        "(bandwidth=150)";
    char *reservation_handle;

    error = globus_gara_reservation_create(gatekeeper_contact,
                                           request_rsl,
                                           &reservation_handle);
```

Table A.1: RSL attributes that can be specified when creating a reservation.

Attribute	Units	Default	Required?	Description
Universal Attributes				
reservation-type			Y	Allowable values: “network”, ‘“compute”, or “disk”
start-time	secs		Y	“now” or time since 00:00:00 UTC January 1, 1970
duration	secs		Y	Duration of reservations
Compute Resource Attributes				
percent-cpu	%	20		Percentage of the CPU’s time given to the reserved process
number-of-nodes		1		Number of nodes needed.
Network Resource Attributes				
endpoint-a			Y	The machine at one end of the network flow. The address must be specified as a dotted IP address, such as 140.221.48.162.
endpoint-b			Y	The machine at the other end of the network flow. The address must be specified as a dotted IP address, such as 140.221.48.162.
bandwidth	Kbps	8		How fast a flow can transfer data
directionality		unidirectional-ab		Currently not implemented, but this would allow bidirectional reservations
Disk Resource Attributes				
size	KB			The storage space needed for a single file.
bandwidth	Kbps	8		How fast data can be read from or written to a file.

Note that the gatekeeper contact is a string obtained from another location, such as the MDS. An example gatekeeper contact may look like:

```
dslnet2.mcs.anl.gov:754:/C=US/O=Globus/O=Argonne
National Laboratory/OU=Mathematics and Computer
Science Division/CN=dslnet2.mcs.anl.gov
```

For more information on the gatekeeper and gatekeeper contacts, see http://www.globus.org, and read about GRAM.

A.1.4 Modifying a reservation

Modifying a reservation is similar to creating a reservation, except that instead of providing a gatekeeper contact, you provide the handle to the reservation that you created earlier:

```
    int  error;
    char *request_rsl = "&(reservation-type=network)"
                        "(start-time=953158862) (duration=3600)"
                        "(endpoint-a=140.221.48.146)"
                        "(endpoint-b=140.221.48.106)"
                        "(bandwidth=150)";
    char *new_reservation_handle;

    error = globus_gara_reservation_create(reservation_handle,
                                           request_rsl,
                                           &new_reservation_handle);
```

A.1.5 Querying a reservation

If you would like to find out the status of a reservation, you can query it:

```
    int error;
    int status;

    error = globus_gara_reservation_status(reservation_handle,
                                           &status);
```

If there is not an error, the status will be one of:

```
GLOBUS_GARA_RESERVATION_STATUS_NOT_STARTED
GLOBUS_GARA_RESERVATION_STATUS_NOT_STARTED_BOUND
GLOBUS_GARA_RESERVATION_STATUS_READY_NOT_BOUND
GLOBUS_GARA_RESERVATION_STATUS_ACTIVE
GLOBUS_GARA_RESERVATION_STATUS_FINISHED
```

A reservation is bound if a previous call to `globus_gara_reservation_bind` succeeded. A reservation is ready if the current time is later than the start time, and the duration has not yet elapsed. A reservation is active if it is both ready and bound. A reservation is finished if the current time is later than the start time plus the duration.

A.1.6 Binding a reservation

When you are ready to use a reservation, you need to bind it in order to begin using the reservation:

```
int   error;
char  *bind_parameters = "&(which-endpoint=a)
                           (endpoint-a-port=1234)
                           (endpoint-b-port=5678)";

error = globus_gara_reservation_bind(reservation_handle,
                                     &bind_parameters);
```

Notice that the run-time parameters are specified as an RSL string. Currently, bind parameters are only specified for compute and network reservations. For compute reservations that use percent-cpu, the only parameter to be specified is process-id, which specifies the process ID of the process that will be receiving the reservation. For network reservations, there are three parameters:

- *which-endpoint:* If the reservation is being bound from a machine involved in the reservation, this specifies which machine it is. The machine is either “a” or “b”, and it matches what was specified in the reservation request. If a different machine is binding the reservation on behalf of the processes involved, simply use “a”.

- *endpoint-a-port:* This is the port used by endpoint-a, as specified in the reservation request. Because the current GARA implementation assumes that data is being sent from endpoint-a to endpoint-b, this will be the port used by the sender.
- *endpoint-b-port:* This is the port used by endpoint-b, as specified in the reservation request. Because the current GARA implementation assumes that data is being sent from endpoint-a to endpoint-b, this will be the port used by the receiver.

Note that a reservation is not considered active until it is bound. Once a reservation has both begun and been bound, then GARA does whatever setup is necessary in order to ensure that the reservation is granted. It is okay if the reservation is bound before it has begun, because GARA will wait to automatically enable the reservation once it begins.

If you will not be using a reservation temporarily but you will resume using it before it has expired, you can unbind the reservation:

```
int   error;

error = globus_gara_reservation_unbind(reservation_handle);
```

Unbinding a reservation may simplify operations for the underlying resource, or may allow it more flexibility. Once you unbind a reservation, you may bind it again.

A.1.7 Using callbacks

If you would like to be informed whenever the status of a reservation changes (see Section A.1.5 above), you can use a callback function. Once you register a callback function, it will immediately be called once, to provide the current status, and will be called every time the status changes afterwards. Note that more types of status can be provided to the callbacks than to the `globus_gara_reservation_status()` function. For a complete list of the different statuses that can be provided to a callback function, see the list in Section A.2.1.

First you need to create a callback function:

```
static void callback_handler(
    char                              *reservation_handle,
    globus_gara_reservation_event_t   event,
    void                              *user_parameter)
{
    /* Place code here to examine the event */
    /* If it is a status event, event.event_type will be */
    /* GLOBUS_GARA_STATUS_EVENT, and the status will be in  */
    /* event.event. */
    if (event.event == GLOBUS_GARA_STATUS_EVENT)
    {
        if (event.event_type == GLOBUS_GARA_RESERVATION_STATUS_FINISHED)
        {
            /* React to reservation being finished */
        }
    }
    return;
}
```

Then you need to register this function with GARA. You need to register the function for each reservation that you wish to monitor:

```
    int error;

    error = globus_gara_reservation_callback_register(
                reservation_handle,
                callback_handler, /* pointer to above function */
                NULL);
```

Note that the last parameter you pass to the registration function will be provided to your callback function as the `user_parameter`.

If you would no longer like to have a function called when the status changes, you can unregister it:

```
int error;

error = globus_gara_reservation_callback_remove(
            reservation_handle,
            callback_handler);
```

Note that you can register multiple callback functions for a single reservation handle.

A.1.8 Canceling a reservation

When you are finished using a reservation you should cancel it, using the reservation handle that you obtained when you created the reservation.

```
globus_gara_reservation_cancel(reservation_handle);
```

When you cancel a reservation, all of the callbacks that have been registered for that reservation will automatically be cancelled.

A.1.9 Deactivating GARA

When you have finished using GARA you should deactivate it, to allow GARA to clean up:

```
globus_module_deactivate(GLOBUS_GARA_CLIENT_MODULE);
```

A.2 GARA reference

A.2.1 Constants

This section describes the constants used by the GARA API. You will find them all either in “globus_gara_client.h” or in “globus_gara_common.h”. Note, however, that “globus_gara_client.h” includes “globus_gara_common.h” for you.

Errors

`GLOBUS_GARA_ERROR_NONE:`

No error has occurred.

`GLOBUS_GARA_ERROR_UNKNOWN:`

An error has occurred, but GARA just doesn't know what it is.

`GLOBUS_GARA_ERROR_MODULE_NOT_ACTIVE:`

You have tried to use GARA without activating the module first.

`GLOBUS_GARA_ERROR_BAD_PARAMETER:`

A bad parameter, such as a NULL reservation handle, has been passed to a GARA function.

`GLOBUS_GARA_ERROR_ZERO_LENGTH_RSL:`

An RSL string was provided, but it is empty. It may be that this is never returned.

`GLOBUS_GARA_ERROR_BAD_RSL:`

There is an error, probably a syntax error, in the RSL string.

`GLOBUS_GARA_ERROR_BAD_RESERVATION_HANDLE:`

The reservation handle that was provided isn't really a reservation handle.

`GLOBUS_GARA_ERROR_CONNECTION_FAILED:`

GARA was unable to connect to the gatekeeper.

`GLOBUS_GARA_ERROR_AUTHORIZATION:`

GARA was unable to authorize with the gatekeeper. Did you run grid-proxy-init?

`GLOBUS_GARA_ERROR_GATEKEEPER_MISCONFIGURED:`

This is reported when there is a serious error in the setup of the gatekeeper. Talk to your system administrator.

`GLOBUS_GARA_ERROR_VERSION_MISMATCH:`

This is reported when you are contacting an old gatekeeper. You will need to upgrade to a newer gatekeeper to use GARA.

`GLOBUS_GARA_ERROR_UNKNOWN_RESERVATION_TYPE:`

The reservation type in the RSL reservation request must be one of “network”, “compute”, or “disk”, but it was not.

`GLOBUS_GARA_ERROR_PROTOCOL_FAILED:`

There was a problem communicating with the gatekeeper.

`GLOBUS_GARA_ERROR_MISSING_RESERVATION_TYPE:`

The reservation type in the RSL reservation request was not provided.

`GLOBUS_GARA_ERROR_OUT_OF_MEMORY:`

A request for memory failed. You are in trouble!

`GLOBUS_GARA_ERROR_MISSING_ENDPOINT_A:`

A network reservation request didn’t specify endpoint-a.

`GLOBUS_GARA_ERROR_MISSING_ENDPOINT_B:`

A network reservation request didn’t specify endpoint-b.

`GLOBUS_GARA_ERROR_CANT_MAKE_RESERVATION:`

The reservation cannot be made. Probably there are other reservations already at the same time, and there is no room for your reservation.

`GLOBUS_GARA_ERROR_PROBLEM_WITH_LRAM:`

The most likely cause of this error is that the resource manager is not running or that communication with it has failed.

`GLOBUS_GARA_ERROR_HTTP_UNPACK_FAILED:`

A serious protocol error happened, probably a programming error on our part, not yours.

`GLOBUS_GARA_ERROR_BAD_RESERVATION_OBJECT:`

This error probably means that you tried to make a network reservation for an endpoint that the resource manager has not been configured to allow reservations for.

`GLOBUS_GARA_ERROR_GARA_SERVICE_EXECUTABLE_NOT_FOUND:`

The gatekeeper is misconfigured. In particular, it cannot find the `globus_gatekeeper_gara_service` executable.

`GLOBUS_GARA_ERROR_CANT_CONTACT_RESOURCE_MANAGER:`

The resource manager is unavailable. Check to make sure that it is running.

`GLOBUS_GARA_ERROR_UNKNOWN_GRAM_ERROR:`

Some error in the underlying "GRAM Gatekeeper" protocol has failed.

Callback and Status Constants

The following events are reported to callbacks:

`GLOBUS_GARA_STATUS_EVENT:`

The status of the reservation has changed. See the lists of status constants below.

`GLOBUS_GARA_CHANGE_EVENT:`

The reservation has been preempted, or the reservation quantity (like bandwidth) has changed. See the list of changes below.

`GLOBUS_GARA_MONITOR_EVENT:`

The reservation is apparently trying to use more than what it reserved. For network reservations, the percent of packets that conformed to the reservation are reported in the `quantity` parameter of the callback event structure (Section A.2.2). For more information on how this can be used, see Section 7.2.1.

The following statuses can be reported to callbacks on a status event or in response to a user calling globus_gara_reservation_status.

`GLOBUS_GARA_RESERVATION_STATUS_NOT_STARTED:`

The reservation has not yet begun (the current time is before the start time).

`GLOBUS_GARA_RESERVATION_STATUS_NOT_STARTED_BOUND:`

Although the reservation has not yet begun, the reservation has been bound.

`GLOBUS_GARA_RESERVATION_STATUS_READY_NOT_BOUND:`
: The reservation has begun (the current time is after the start time) but cannot yet be used because it has not been bound yet.

`GLOBUS_GARA_RESERVATION_STATUS_ACTIVE:`
: The reservation has begun and been bound.

`GLOBUS_GARA_RESERVATION_STATUS_FINISHED:`
: The reservation is over. That is, the current time is greater than the start time plus the duration of the reservation.

The following changes can be reported on a CHANGE_EVENT:

`GLOBUS_GARA_RESERVATION_CHANGE_PREEMPTED:`
: The reservation has been preempted because a more important reservation has occurred. Currently, this will not be reported, because preemption has not yet been implemented.

`GLOBUS_GARA_RESERVATION_CHANGE_QUANTITY:`
: The quantity (like bandwidth) has been changed. This occurs for bulk transfer reservations, as described in Section 7.2.2.

A.2.2 Data structures

This is a description of the data structures used by GARA.

The Event Data Structure

```
typedef struct
{
  int     event_type;
  int     event;
  double  quantity;
} globus_gara_reservation_event_t;
```

This structure is provided to callback functions. The event type and event are constants from the list above. The quantity is provided when the event is a change event indicating that the quantity has changed.

Callback functions

```
typedef void (*globus_gara_reservation_callback_t)(
    char                              *reservation_handle,
    globus_gara_reservation_event_t  event,
    void                              *user_parameter);
```

This is the type of function that must be used for callback functions.

A.2.3 Functions

Note that all of the functions in GARA return an integer. This integer is the error code, if any error occurred. See the list of errors in Section A.2.1.

globus_gara_reservation_create

```
int globus_gara_reservation_create(
    const char *manager_contact,
    const char *reservation_specification,
    char       **reservation_handle);
```

Description: This function attempts to make a reservation.

manager_contact: *(IN)* The contact string for the gatekeeper that controls access to the resource manager for the resource you wish to make a reservation with.

reservation_specification: *(IN)* An RSL string describing the attributes you wish to have for your reservation. See Section A.1.2 above.

reservation_handle: *(OUT)* If the reservation was successfully made, a pointer to your reservation handle will be provided in this parameter. The memory for the reservation handle is allocated by `globus_malloc()`, and it is your responsibility to free the memory with `globus_free()` when you are done.

globus_gara_reservation_modify

```
int globus_gara_reservation_modify(
    const char *old_reservation_handle,
    const char *reservation_specification,
    char       **new_reservation_handle);
```

Description: This function attempts to modify a new reservation. Note that if the reservation is changed, you are provided with a new reservation handle. While current versions of GARA will provide an identical reservation handle, future versions of GARA may not, therefore we always return a new reservation handle.

old_reservation_handle: *(IN)* The handle for the reservation that you wish to modify.

reservation_specification: *(IN)* An RSL string describing the new attributes you wish to have for your reservation. See Section A.1.2 above.

new_reservation_handle: *(OUT)* If the reservation was successfully modified, a pointer to your reservation handle will be provided in this parameter. The memory for the reservation handle is allocated by `globus_malloc()`, and it is your responsibility to free the memory with `globus_free()` when you are done.

globus_gara_reservation_bind

```
int globus_gara_reservation_bind(
    const char *reservation_handle,
    const char *bind_parameters);
```

Description: This claims a reservation by providing run-time parameters.

reservation_handle: *(IN)* The handle for the reservation that you wish to bind.

bind_parameters: *(IN)* An RSL string describing the new attributes you wish to have for your reservation. See Section A.1.6 above.

globus_gara_reservation_unbind

```
int globus_gara_reservation_unbind(
    const char *reservation_handle);
```

Description: This “un-claims” a reservation. The reservation is still valid and can be used again by calling `globus_gara_reservation_bind()` again.

reservation_handle: *(IN)* The handle for the reservation that you wish to bind.

globus_gara_reservation_status

```
int globus_gara_reservation_status(
    const char *reservation_handle,
    int        *status;
```

reservation_handle: *(IN)* The handle for the reservation that you wish to query.

status: *(OUT)* The status of the reservation. It is one of the constants described in Callback and Status Constants.

globus_gara_reservation_callback_register

```
int globus_gara_reservation_callback_register(
    const char                        *reservation_handle,
    globus_gara_reservation_callback_t  callback_function,
    void                               *user_parameter);
```

Description: After this function successfully completes, the specified callback function will be called whenever the status of a reservation changes. It will also be immediately called once to provide the current status of the reservation. Note that multiple callbacks can be registered for a single reservation.

reservation_handle: *(IN)* The handle for the reservation for which you wish to receive callbacks.

callback_function: *(IN)* The function that will be called by GARA when the status of a reservation changes.

user_parameter: *(IN)* The value you provide here will be provided to the callback function unmodified.

globus_gara_reservation_callback_remove

```
int globus_gara_reservation_callback_remove(
    const char                           *reservation_handle,
    globus_gara_reservation_callback_t  callback_function);
```

Description: After this function successfully completes, the specified callback function will no longer be called when the status of the reservation changes.

reservation_handle: *(IN)* The handle for the reservation for which you wish to receive callbacks.

callback_handle: *(IN)* The function that will be called by GARA when the status of a reservation changes.

globus_gara_reservation_cancel

```
int globus_gara_reservation_cancel(
    const char *reservation_handle);
```

Description: This cancels a reservation. When a reservation is cancelled, the reservation handle (and copies of it) may not be used anymore. For example, if you try to bind the cancelled reservation, it will fail.

reservation_handle: *(IN)* The handle for the reservation that you wish to cancel.

globus_gara_version

```
int globus_gara_version(void);
```

Description: This returns the current version number for GARA.

globus_gara_reservation_client_debug

```
int globus_gara_client_debug(void);
```

Description: This enables debugging mode. Output will be printed to stderr.

globus_gara_client_error_string

```
const char *globus_gara_client_error_string(
    int error_code);
```

Description: For any error code returned by GARA, this provides a printable string that corresponds to the error code.

error_code: *(IN)* The error code for which you wish to obtain a string representation.

A.3 Example program using GARA

This code has been greatly simplified but should give you the basic idea of how to use GARA.

```
#include "globus_gara_client.h"

int reservation_ready = GLOBUS_FALSE;

static void callback_handler(
    char                               *reservation_handle,
    globus_gara_reservation_event_t  event,
    void                               *user_parameter);

int main(int argc, char **argv)
{
  int           error;
  int           seconds;
  int           test_index;
  char          *reservation_handle;
  char          *reservation_rsl_string;

  /* Initialize */
  globus_module_activate(GLOBUS_GARA_CLIENT_MODULE);

  /* Make a reservation that starts in 20
   * seconds and goes for an hour
   * Of course, many of the RSL parameters would not
   * actually be static in a real program. */
  reservation_rsl_string = globus_malloc(256);
  sprintf(reservation_rsl_string,
          "&(reservation-type=network)(start-time=%d)
            (duration=%d) (endpoint-a=%s)
            (endpoint-b=%s) (bandwidth=%d) (protocol=tcp)",
          (int) time(NULL) + 20, 3600,
          "128.135.11.1", "128.135.11.6", 150);
  error = globus_gara_reservation_create(
              parameters.gatekeeper_contact,
              reservation_rsl_string, &reservation_handle);

  /* Set up a callback to let us know when the
   * reservation is ready. */
  error = globus_gara_reservation_callback_register(
              reservation_handle,
              callback_handler, NULL);

  /* Wait for the reservation to become active */
  while (!reservation_ready)
    sleep(1);

  /* Bind the reservation */
  error = globus_gara_reservation_bind(reservation_handle,
              "&(which-endpoint=a)
                (endpoint-a-port=9999)
                (endpoint-b-port=9999)");

  /* Use the reservation... */
  ;

  /* Remove the callback */
  error = globus_gara_reservation_callback_remove(
              reservation_handle,
              callback_handler);

  /* Cancel the reservation */
  error = globus_gara_reservation_cancel(reservation_handle);
```

```
  /* Clean up */
  globus_free(reservation_rsl_string);
  globus_free(reservation_handle);
  globus_module_deactivate(GLOBUS_GARA_CLIENT_MODULE);
  return 0;
}

static void callback_handler(
    char                             *reservation_handle,
    globus_gara_reservation_event_t  event,
    void                             *user_parameter)
{
  if (event.event_type==GLOBUS_GARA_STATUS_EVENT
    && event.event==GLOBUS_GARA_RESERVATION_STATUS_READY_NOT_BOUND)
  {
    reservation_ready = GLOBUS_TRUE;
  }
}
```

REFERENCES

[1] B. Allcock, J. Bester, A.L. Chervenak, I. Foster, C. Kesselman, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, April 2001.

[2] S. Blake, D. Black, M. Carlson, M. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. *Internet RFC 2475*, December 1998.

[3] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the internet architecture: an overview. *Internet RFC 1633*, July 1994.

[4] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP)-version 1 functional specification. *Internet RFC 2205*, September 1997.

[5] Eugene Burger. *The Experience of Magic*. Kaufman and Company, 1989.

[6] Andrew T. Campbell. *A Quality of Service Architecture*. PhD thesis, Lancaster University, England, January 1996.

[7] CCITT. Recommendation X.509: The directory – authentication framework. Technical report, 1988.

[8] P.F. Chimento et al. QBone bandwidth broker architecture. Work in Progress, available from `http://sss.advanced.org/bb/`.

[9] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*. IEEE Computer Society Press, 1999.

[10] D. Comer. *Internetworking with TCP/IP*. Prentice-Hall International Editions, 1988.

[11] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[12] Tom DeFanti and Rick Stevens. Teleimmersion. In *[21]*, pages 131–156. Morgan Kaufmann Publishers, 1999.

[13] A. Demir, October 2000. Personal communication with Alper Demir of ISI.

[14] D. Ferraiolo, J. Barkley, and R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):33–64, February 1999.

[15] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[16] D. Ferrari and D. Verma. Resource partitioning for real-time communication. In *Proceedings of the First IEEE International Symposium on Global Data Networking*, December 1993.

[17] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, 1997.

[18] J. Forgie. ST - a proposed internet stream protocol. *Internet IEN 119*, September 1979.

[19] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[20] I. Foster and C. Kesselman. Globus: A toolkit-based grid architecture. In *[21]*, pages 259–278. Morgan Kaufmann Publishers, 1999.

[21] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

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

[23] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[24] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *International Workshop on Quality of Service*, pages 181–188, 2000.

[25] I. Foster, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, 1999. `http://www.mcs.anl.gov/qos`.

[26] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Nitzberg, W. Saphir, and Marc Snir. *MPI–The Complete Reference. Volume 2–The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[27] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[28] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[29] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1998.

[30] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding PHB group. *Internet RFC 2597*, June 1999.

[31] G. Hoo, W. Johnston, I. Foster, and A. Roy. QoS as middleware: Bandwidth broker system design. Technical report, LBNL, 1999.

[32] K. Jackson, November 2000. Personal communication with Keith Jackson of Lawrence Berkeley Laboratory.

[33] V. Jacobson, K. Nichols, and K. Poduri. An expedited forwarding PHB. *Internet RFC 2598*, June 1999.

[34] T. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgement channel: A study of TCP/IP performance. In *Proceedings of the IEEE INFOCOM*. 1997.

[35] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QOS in Real-Time Mach. *In the Proceedings of Multimedia Japan 96*, April 1996.

[36] J. Linn. Generic security service application program interface. *Internet RFC 1508*, 1993.

[37] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini multimedia storage server. In S. M. Chung, editor, *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.

[38] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. In *Proceedings of ACM SIGCOMM, volume 27, number 3*. 1997.

[39] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[40] K. Nahrstedt and J. M. Smith. The QoS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[41] K. Nahrstedt and J. M. Smith. Design, implementation and experiences of the OMEGA end-point architecture. *IEEE JSAC, Special Issue on Distributed Multimedia Systems and Technology*, 14(7):1263–1279, September 1996.

[42] K. Nahrstedt, D. Wichadakul, and D. Xu. Distributed QoS compilation and runtime instantiation. In *Proceedings of the IEEE/IFIP International Workshop on QoS (IWQoS'2000)*, Pittsburgh, June 2000.

[43] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. *Internet RFC 2747*, December 1998.

[44] K. Nichols, V. Jacobson, and L. Zhang. A two-bit differentiated services architecture for the Internet. *Internet RFC 2638*, July 1999.

[45] R. Nitzan and B. Tierney. Experiences with TCP/IP over an ATM OC12 WAN. Technical report, LBNL, April 1999.

[46] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive control of distributed applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[47] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. *Internet RFC 3031*, January 2001.

[48] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality of service for message passing programs. In *Proceedings of the IEEE/ACM SC2000 Conference*, November 2000.

[49] V. Sander, W. Adamson, I. Foster, and A. Roy. End-to-end provision of policy information for network QoS. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[50] V. Sander, I. Foster, and A. Roy. Implementing a premium service based on the expedited forwarding per-hop behavior. Technical report, Argonne National Laboratory, September 2000.

[51] V. Sander, I. Foster, A. Roy, and L. Winkler. A differentiated services implementation for high-performance TCP flows. *The International Journal of Computer and Telecommunications Networking*, 34:915–929, 2000.

[52] J. Steiner, B. C. Neuman, and J. Schiller. Kerberos: An authentication system for open network systems. In *Usenix Conference Proceedings*, pages 191–202. 1988.

[53] W. Stevens. *TCP/IP Illustrated, Vol. 1 The Protocols*. Addison-Wesley, 1997.

[54] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc., 1992.

[55] B. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, and F. Drake. A network-aware distributed storage cache for data intensive environments. In *Proceedings of IEEE High Performance Distributed Computing conference (HPDC-8)*, August 1999.

[56] M. Wahl, T. Howes, and S. Kille. RFC 2251: Lightweight directory access protocol (v3). *Internet RFC 760*, 1997.

[57] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, Portland, Oregon, 1997. IEEE Press.

[58] D. Xu, K. Nahrstedt, and A. Viswanathan D. Wichadakul. QoS and contention-aware multi-resource reservation. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, August 2000.

[59] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, January 1997.