MPICH-GQ: Quality-of-Service for Message Passing Programs

Alain Roy [*] Ian Foster [*†] William Gropp [†] Nicholas Karonis [‡] Volker Sander [§] Brian Toonen [†]

Abstract

Parallel programmers typically assume that all resources required for a program's execution are dedicated to that purpose. However, in local and wide area networks, contention for shared networks, CPUs, and I/O systems can result in significant variations in availability, with consequent adverse effects on overall performance. We describe a new message-passing architecture, MPICH-GQ, that uses quality of service (QoS) mechanisms to manage contention and hence improve performance of message passing interface (MPI) applications. MPICH-GQ combines new QoS specification, traffic shaping, QoS reservation, and QoS implementation techniques to deliver QoS capabilities to the high-bandwidth bursty flows, complex structures, and reliable protocols used in high-performance applications—characteristics very different from the low-bandwidth, constant bit-rate media flows and unreliable protocols for which QoS mechanisms were designed. Results obtained on a differentiated services testbed demonstrate our ability to maintain application performance in the face of heavy network contention.

Keywords: MPI, Quality of Service, Differentiated Services, TCP

1 Introduction

The performance achieved by parallel programs is often adversely affected by contention for resources such as networks and I/O systems. In the case of networks, even a small amount of contention over a critical link can play havoc with overall performance, particularly when an application is using TCP/IP as its communication protocol: TCP/IP's built in contention-avoidance mechanisms can reduce communication rates drastically, potentially idling many processors.

[*]Department of Computer Science, The University of Chicago, Chicago, IL 60637, U.S.A.

[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

[‡]High-Performance Computing Laboratory, Department of Computer Science, Northern Illinois University, DeKalb, IL 60115, U.S.A.

[§]Central Institute for Applied Mathematics, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany

One approach to dealing with contention is to explicitly manage the allocation of scarce resources to different purposes. If appropriate mechanisms can be provided for expressing application requirements, for arbitrating between different requirements, for enforcing allocations, and for providing feedback to applications concerning achieved performance, then applications can, in principle, adapt their behavior according to resource availability. However, while such quality of service (QoS) mechanisms have been developed for low-bandwidth, constant bit-rate media flows with unreliable protocols [20, 4, 27, 29, 28], the high-performance, often bursty traffic patterns, complex communication structures, and reliable protocols encountered in high-performance computing applications pose new challenges. Furthermore, the sockets-based application programming interfaces (APIs) typically provided for managing QoS are not appropriate for scientific applications.

We describe here a system, MPICH-GQ, that addresses the problems just listed, providing the parallel programmer with a QoS framework that supports

- high-bandwidth (10s of Megabits per second: Mb/s) flows;
- reliable protocols, specifically TCP/IP;
- a variety of different low-level QoS enforcement mechanisms;
- both immediate and advance reservation, and co-reservation, of CPU, network, and other resources needed for end-to-end performance; and
- a familiar high-level programming model, namely the message passing interface (MPI).

Our prototype MPICH-GQ implementation combines elements of the MPICH-G2 (formerly MPICH-G) wide area implementation of MPI [17, 10] and the General-purpose Architecture for Reservation and Allocation (GARA) QoS framework [14]. Experimental studies with simple MPI benchmark studies demonstrate our ability to deliver high performance in the face of network contention.

This work is part of a larger project focused on exploiting MPI as a standards-based API for advanced network computing. This work has produced new techniques for constructing topology-aware collective operations [23] as well as a first version of MPICH-G [10], a “Grid-” [12] or network-aware implementation of MPI that uses mechanisms provided by the Globus toolkit [11] to address security, startup, remote I/O, and other issues that can hinder distributed execution. The PACX [15], MetaMPI [7], and MAGPIE [24] systems also address some of these issues, but not QoS.

2 Quality of Service: A Brief Review

We review briefly the state of the art in network QoS mechanisms, focusing on methods used within Internet Protocol (IP)-based packet-switched networks.

In today’s Internet, the packets constituting an application-level flow pass through a series of routers as they travel from their source to their destination. An individual router must make decisions with implications for perceived QoS whenever more than one incoming packet can be forwarded at the same time. The router must then choose to forward one packet before the other(s), which are queued and hence delayed; if the number of incoming packets exceeds the output rate of the router for an extended period, the router must also choose which packets to discard.

Two primary approaches to QoS have been proposed within the IP context. In the Integrated Services (IS) [2] approach, a reservation is made at each router between the endpoints of a reservation, usually via the Resource ReserVation Protocol (RSVP) [3]. Each router distinguishes each of the reserved flows and provides each flow with performance guarantee, either statistical or strict. The IS approach has been criticized as being too “heavy,” however, because each router is required to recognize and treat each application-level flow separately, which may be too much of a burden to place on routers in the core of a large network such as the Internet.

In the Differentiated Services (DS) [1] approach, routers that are at the “edge” of a DS network recognize packets that should receive better service by classifying the packets based on information in the header, such as source and destination addresses and ports. For example, a network provider may choose to give better-quality service to all packets sent in video flows, or may have a bandwidth reservation system that allows applications to request specific amounts of bandwidth for particular application flows. Once an edge router classifies a packet as needing better service, it marks that packet in the header with a particular service. In the interior of the network, packets are no longer fully classified as they were by the edge routers, but they are treated as an aggregate based on the service marked in the packet. This approach greatly simplifies the task of the routers in the interior of the network.

In addition to classifying and marking the packets, edge devices may have to perform policing and shaping. *Policing* is a mechanism to ensure that senders do not send too much high-quality traffic; if the sender transmits data too quickly, policing will throw out traffic above a certain rate. Policing is often implemented through a token bucket mechanism. The size of the token bucket controls how quickly an application can send data: tokens are gradually added to the token bucket and packets are only sent if there are tokens in the bucket. Policing is important when the network provider wishes to offer not simply different types of service, but some sort of guarantee on a service: if access to the service is unrestricted, it may be impossible to provide such guarantees. *Shaping* is important when application traffic is bursty. If these bursts are not smoothed to be less bursty, policing may cause packets to be dropped. As we explain below, shaping can be performed either in the router or in the application.

The Internet Engineering Task Force (IETF) has defined two different types of DS services. These are not services in the “end-to-end” sense of the word, but instead are *per hop behaviors* (PHBs). That is, they define how packets are treated at each router. One PHB that is of particular interest to us in the *Expedited Forwarding* (EF) behavior which says that all packets in the “expedited” router queue are sent before any other packets are sent. Clearly, to prevent starvation of nonexpedited flows, the number of expedited packets must be carefully limited. It is possible to build a premium end-to-end service that provides statistical bandwidth guarantees on top of the EF PHB by doing careful admission control and policing at the edge routers. Normally, admission control is performed not by the router but by an external QoS system, usually referred to as a *bandwidth broker*. In this paper, we will not discuss admission control and the bandwidth broker in detail, although it is part of the GARA system as described below.

In brief, then, the task of providing application-level QoS maps to that of configuring key parameters (flow rates, bucket sizes) within individual routers. This mapping is fairly straightforward for the media applications that have motivated most work on QoS, because of their simple and regular communication structures. For example, an audio stream may comprise 1000 bit packets, generated every 64th of a second, for a total bandwidth requirement of 64 Kb/s. Therefore, we configure the underlying network to support a flow with premium bandwidth of 64 Kb/s and a token bucket depth of 1000 bits. As we discuss in the following, things are more complex for high-performance applications.

3 Quality of Service and MPI

The communication structures associated with MPI applications are often significantly more complex than in media applications, 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 (on a LAN or WAN) TCP/IP. 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, we 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 kicks 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 1 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.

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 interactions.

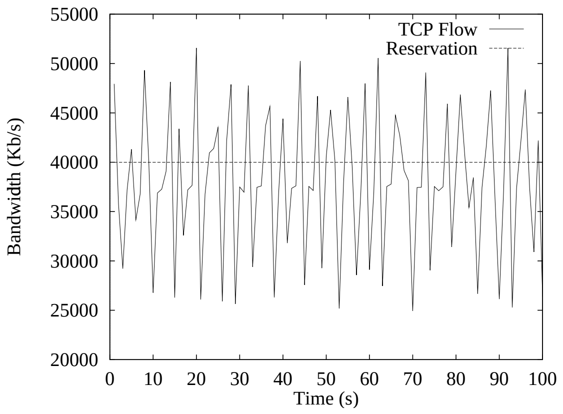


Figure 1: An application using TCP has made a reservation for only 40 Mb/s, when it is sending at 50M b/s.

4 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. Our initial work focuses on IP networks and DS mechanisms, but we believe that the basic principles apply in other contexts, for example, within a parallel computer with an interconnect that supports QoS mechanisms.

Figure 2 shows the principal components of the MPICH-GQ architecture. These are:

- The *MPICH implementation of MPI* [17] 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 in actuality the results presented used a slightly 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* [10] provides low-level security, startup, and other functions for wide area networks.
- The *globus-io 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* [13] 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.

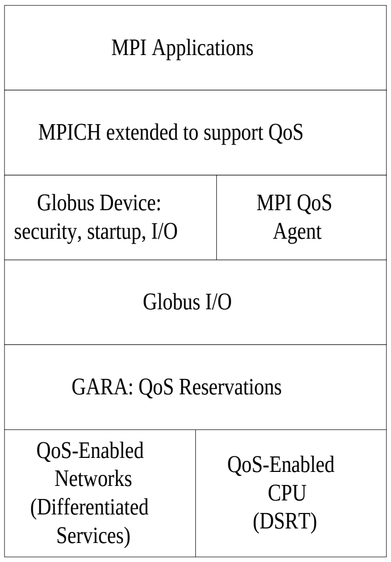

MPI Applications
MPICH extended to support QoS
Globus Device: security, startup, I/O
MPI QoS Agent
Globus I/O
GARA: QoS Reservations
QoS-Enabled Networks (Differentiated Services)
QoS-Enabled CPU (DSRT)

Figure 2: 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.

We now proceed to describe the MPI attribute extensions, GARA architecture, and the techniques used to deal with TCP flows.

4.1 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 [18, 16, 19]).

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 implementors 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 quite low-level specifications of required QoS to the underlying QoS system. A typical specification is illustrated in Figure 3. The QoS class may be “best-effort” (i.e., no QoS), “low-latency” (suitable for small message traffic [32, 31]: 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.

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.

Note that, in our prototype, we do not yet fully support

```
struct qos_attribute
{
  u_int32_t qosclass;
  double    bandwidth; /* Peak bandwidth in kbps */
  int       max_message_size; /* Max size used in MPI_Send */
} QoS, *Qos_p;
    ...
MPI_Attr_put( comm, MPICH_ATM_QOS, &QoS);
MPI_Attr_get( comm, MPICH_ATM_QOS, &Qos_p, &flag );
```

Figure 3: QoS-enhanced MPI code to set and then check the QoS parameters associated with a communicator.

the setting of QoS chracteristics directly from within MPI as this requires modifications to our security mechanisms. The modifications, while not significant, did hold up the implementation.

4.2 The GARA Architecture

An MPI QoS Agent must be able to translate application-level QoS requests into reservations for low-level physical resources. We perform these reservations via requests to GARA, a resource management architecture that supports flow-specific QoS specification, secure immediate and advance co-reservation, online monitoring/control, and policy-driven management of a variety of resource types, including networks [13]. Mechanisms provided by the Globus toolkit [11] are used to address resource discovery and security issues when resources span multiple administrative domains.

GARA defines APIs that allows users and applications to manipulate reservations of different resources in uniform ways. For example, essentially the same calls are used to make an immediate or advance reservation of a network or CPU resource. Once a reservation is made, an opaque object called a reservation handle is returned that allows the calling program to modify, cancel, and monitor the reservation. Other functions allow reservations to be monitored by polling or through a callback mechanism in which a user's function is called every time the state of the reservation changes in an interesting way.

MPICH-GQ can use GARA mechanisms to reserve shared resources, such as networks and CPUs, and then to bind specific flows (sockets) and processes to those reservations. In our work to date, we have demonstrated the ability to generate reservations for an MPI application once the application has been started. In the future, we will integrate the reservation process with MPI startup and execution, so that, for example, an MPI program can select from among alternative resources, according to their availability, and adapt execution strategies or change reservations if reservations cannot be satisfied in full or are preempted.

The GARA implementation must provide admission control and reservation enforcement for multiple resources of different types. Because few resources provide reservation capabilities, we have implemented our own resource manager so as to ensure availability of reservation functions. This manager uses a slot table [6, 21] to keep track of reservations and invokes resource-specific operations to enforce reservations. Requests to this resource manager are made via an internal local resource manager API and result in calls to functions that add, modify, or delete slot table entries; timer-based callbacks generate call-outs to resource-specific routines to enable and cancel reservations. Note that only certain elements of this resource manager need to be replaced to instantiate a new resource interface. To date, we have developed resource managers for DS networks, for the Distributed Soft Real-Time (DSRT) CPU scheduler [22], and for the Distributed Parallel Storage System (DPSS) [34], a network storage system; others are under development.

4.3 Support for TCP Flows

An MPICH-GQ call to GARA requesting 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 [33, 5], 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 [26, 25]. In a DS-based system, this means that we need both a large token bucket on the edge router and an accurate reservation value.

The GARA DS module 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 bytes, bandwidth is in bits per second, and "delay" is in seconds. In our local testbed (described in Section 5.1), the delay is quite small, on the order of a millisecond or two. A two millisecond delay would therefore suggest that the depth of the bucket should be $bandwidth/62$. However,to allow for larger bursts in traffic, we currently use $bandwidth/40$. As we explain in Section 5.4 below, this value is not always adequate.

5 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.

5.1 Experimental Setup

Our experimental configuration, illustrated in Figure 4, is a laboratory testbed at Argonne National Laboratory called the Globus Advance Reservation Network Testbed, or GARNET, which is connected to a number of remote sites. GARNET allows controlled experimentation with basic DS mechanisms; the wide area extensions allow for more realistic operation, albeit with a small number of sites.

We use a DS implementation based on Cisco 7500 series routers, which support the EF PHB with the Modular QoS Command line interface (MQC). In detail, we use the following mechanisms to support our DS implementation:

- A packet classifier is used on each router interface to determine the type of service.
- A token bucket mechanism is used on the ingress ports of edge routers to mark and police the flows for which premium bandwidth is required. It is also used on the ingress router of a domain to police the premium aggregate.
- Priority Queuing is used on the egress port of edge routers to support delay-sensitive UDP flows. Priority queueing ensures that all packets associated with reservations are sent before any other packets. When there are no packets in the priority queue, other packets are allowed to use the entire available bandwidth.

Within GARNET, the routers are connected by OC3 ATM connections; across wide area links, they are connected by VCs of varying capacity. End system computers are connected to routers by either switched Fast Ethernet or OC3 connections.

5.2 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 5 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 quite 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.

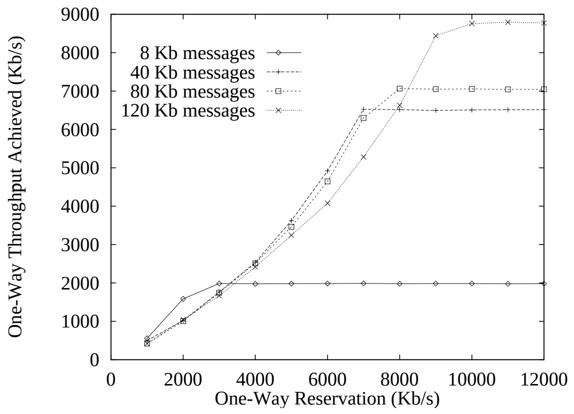


Figure 5: The effect of different reservation sizes for the ping-pong MPICH-GQ program. Each line represents the throughput achieved for a particular message size at different reservation sizes.

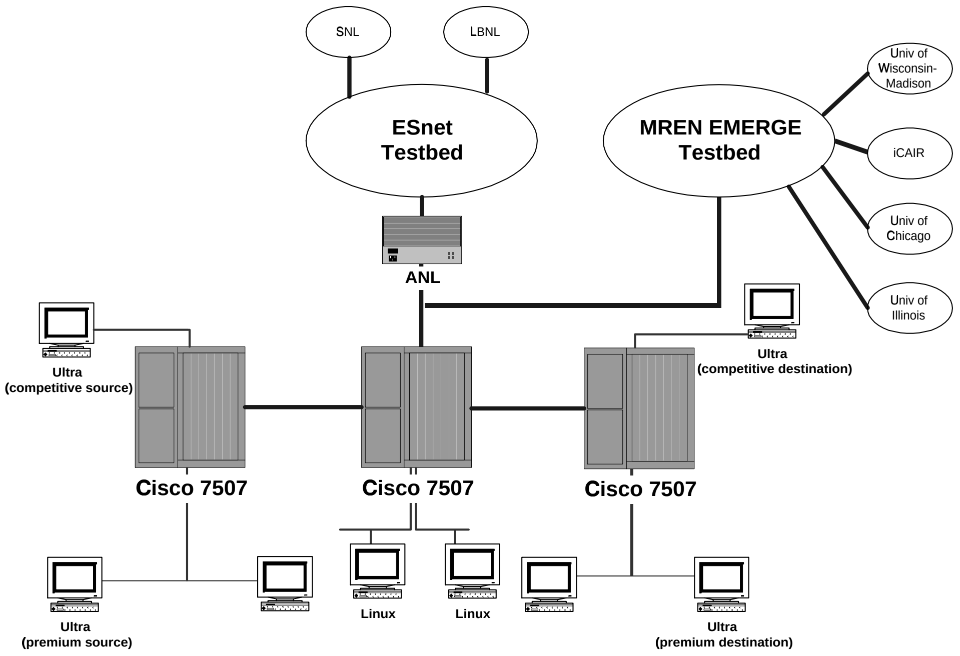


Figure 4: GARNET, our experimental testbed.

5.3 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") and the message size ("frame size") can be adjusted, hence varying both the generated bandwidth and the burstiness of the traffic.

Figure 6 shows the throughput achieved by this program as a function of reservation size for frame sizes of 5, 10, 20, and 30 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.

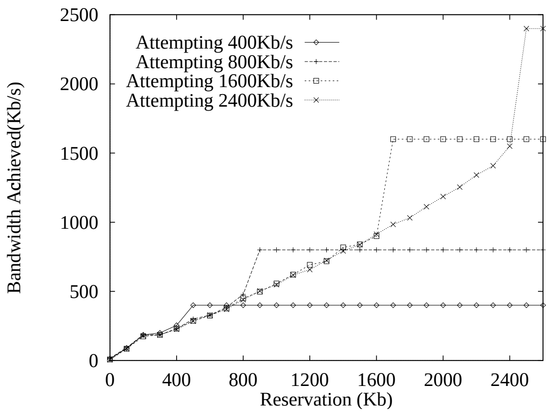


Figure 6: The effect of different reservations on the visualization application attempting different throughputs. Note that making a reservation that is even a little bit too small dramatically decreases the throughput that is achieved.

5.4 The Effect of Burstiness

We outlined in Section 4.3 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 rate, while varying both the burstiness of the traffic (1 frame per second or 10, with the former of course 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 1, demonstrate that there

Table 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.

Bandwidth	Reservation Required		
	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

are limits to the size of the burst that our "normal" token bucket depth can deal with: with the normal depth, the very bursty configurations needs an approximately 50% larger reservation.

Figure 7 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 alll 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 [35]. However, one is also expending scarce system resources. An alternative approach is to incorporate traffic-shaping support into the MPICH-GQ implementation on the end-system.

5.5 Combining Network and CPU Reservations

Up to this point, we have only considered QoS for networks. Unfortunately, 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 [22]. DSRT works by overriding the Unix scheduler and performing soft real-time scheduling of select processes.

Figure 8 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

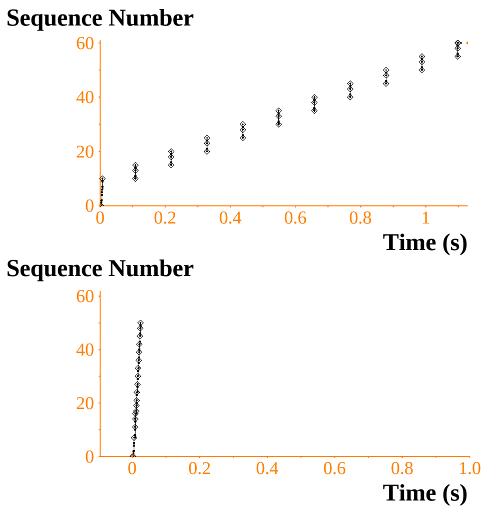
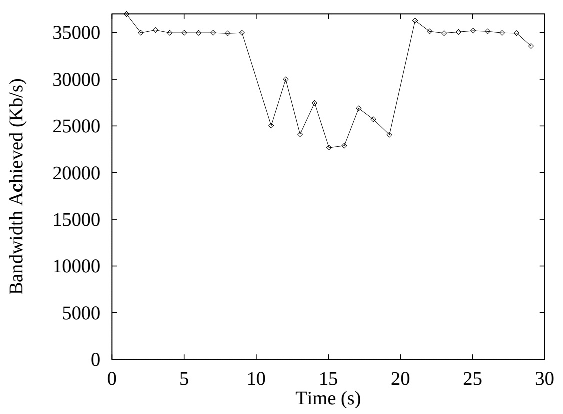


Figure 7: TCP traces of two programs that each send at 400Kb/s, but with very different burstiness characteristics. On the top is a program sending 10 frames per second, and each frame is 40Kb. On the bottom is a program sending just 1 frame per second, and the frame is 400Kb. (This corresponds to the first line of Table 1.) In each case, only one second of the program's execution is shown.

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 affect 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 very little CPU time, and was not significantly affected by the CPU contention. After a modification to make the application do some "work" between sending frames, the application was more affected by the CPU contention.

There are two lessons to draw from this experience.



..

Figure 8: The bandwidth achieved by the visualiation application. Contention for the CPU on the sending side begins at 10 seconds, and a reservation is made at 20 seconds.

First, applications that use TCP and want high performance need careful tuning (such as socket buffer sizes) to actually obtain the high performance. 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)? Applications that have large bandwidths are much more sensitive to CPU contention, and may need CPU reservations to achieve their desired performance.

Figure 9 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 contention. The network congestion begins at time 10 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 acheive 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 clearly 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.

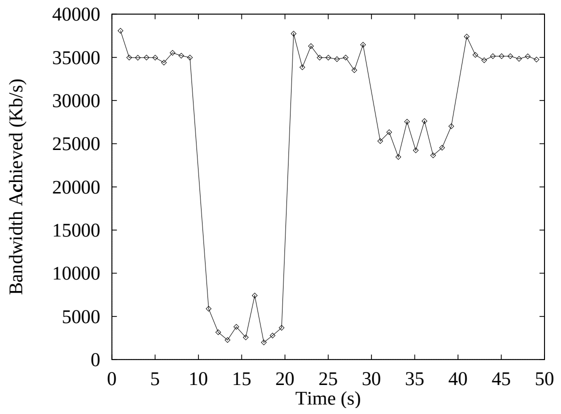


Figure 9: A trace of the bandwidth achieved by the visualization application as it attempts to achieve a constant 35Mb/s rate. Initially it runs well (0-10 seconds), then network congestion affects its bandwidth (11-20 seconds) until a network reservation is made (21-30 seconds). Bandwidth again decreases when there is CPU contention at the sender (31-40 seconds) until there is a CPU reservation (41-50 seconds).

6 Related Work

We review briefly related work on QoS within MPI, the high-level specification of QoS, and QoS for reliable flows.

The only other relevant effort in the context of MPI is work on real-time extensions to MPI. MPI/RT [9] provides a QoS interface but is not an established standard and introduces a new programming interface. Furthermore, the focus is on real-time needs such as predictability of performance and system resource usage more appropriate for embedded systems than for wide area networks.

Other approaches to the high-level specification of QoS include work within the context of CORBA [30, 36] (in the context of embedded systems) and socket libraries [8]. However, these systems are not appropriate for high-performance computing due to their object request broker and socket-based programming models, respectively.

The pan-European research network and its task force Testing Advanced Networking Technologies (TF-TANT) are currently evaluating DS mechanisms for providing QoS to reliable and unreliable flows using commodity hardware, although they are not concerned with higher-level systems like MPI, but with TCP and UDP, at lower bandwidths than considered here.

Acknowledgments

We gratefully acknowledge assistance given by Linda Winkler and Becca Nitzan with the testbed used in these experiments and by Andy Adamson who wrote the UDP traffic generator. John Bresnahan and Joe Link have been instrumental in implementing our QoS extensions to MPI. Numerous discussions with our colleagues Keith Jackson, Gary Hoo, Bill Johnston, Carl Kesselman, and Steven Tuecke have helped shape our approach to quality of service. We also thank Cisco Systems for an equipment donation that allowed the creation of the GARNET testbed. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; by the National Science Foundation; and by the NASA Information Power Grid program.

References

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

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

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

[4] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Resource management for value-added customizable network service. In *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, 1998.

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

[6] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the internet. *ACM/Springer Verlag Journal on Multimedia Systems*, 5(3), 1997.

[7] Thomas Eickermann, Helmut Grund, and Jörg Henrichs. Performance issues of distributed MPI applications in a german gigabit testbed. In *Proc. of the 6th European PVM/MPI Users' Group Meeting*, September 1999.

[8] Microsoft Winsock QoS extensions. `ftp://ftp.microsoft.com/bussys/winsock/winsock2/gqos_spec.doc`.

[9] MPI/RT Forum. `http://www.mpirt.org`.

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

[11] I. Foster and C. Kesselman. Globus: A toolkit-based grid architecture. In *[12]*, pages 259–278.

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

[13] 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.

[14] 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`.

[15] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogenous computing environment. In *EuroPVMMPI'98*, 1998.

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

[17] 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.

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

[19] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill 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.

[20] Roch Guérin and Henning Schulzrinne. Network quality of service. In *[12]*, pages 479–503.

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

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

[23] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *2000 International Parallel and Distributed Processing Symposium (IPDPS '00)*, May 2000.

[24] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, May 1999.

[25] 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.

[26] 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.

[27] A. Mehra, A. Indiresan, and K. Shin. Structuring communication software for quality-of-service guarantees. In *Proc. of 17th Real-Time Systems Symposium*, December 1996.

[28] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware resource management for distributed multimedia applications. *Journal on High-Speed Networking, IOS Press*, December 1998.

[29] 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.

[30] C. O'Ryan, D. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and David L. Levine. Evaluating policies and mechanisms for supporting embedded, real-time applications with corba 3.0. In *Proceedings to the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, June 2000.

[31] 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.

[32] V. Sander, I. Foster, A. Roy, and L. Winkler. A Differentiated Services Implementation for High-Performance TCP Flows. In *Terena Networking Conference 2000 (TNC2000)*. May 2000.

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

[34] B. Tierney, W. Johnston, L. Chen, H. Herzog, G. Hoo, G. Jin, and J. Lee. Distributed parallel data storage systems: A scalable approach to high speed image servers. In *Proc. ACM Multimedia 94*. ACM Press, 1994.

[35] 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.

[36] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for corba objects. In *Theory and Practice of Object Systems*, volume 3, pages 55–73, January 1997.