SCHEDULING TCP IN THE NEMESIS OPERATING SYSTEM*

Thiemo Voigt and Bengt Ahlgren
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
thiemo@sics.se, bengta@sics.se

Abstract

The Nemesis operating system is designed to provide Quality of Service to applications. Nemesis also allows applications to reserve CPU time and transmit bandwidth on network interfaces. We have implemented a TCP for Nemesis that makes use of these guarantees.

We show that the Nemesis transmit scheduler rate-controls TCP traffic and thus leads to predictable traffic behavior when applications choose not to utilize non-allocated bandwidth. Applications that want to make use of the non-allocated transmit bandwidth receive the guaranteed bandwidth plus a share of the non-allocated bandwidth.

We also study the impact of the guaranteed fraction of CPU time on the throughput that networked applications achieve. We measure the amount of CPU time applications have to reserve in order to run the TCP protocol stack and send data at a particular speed. We show that these values hold even when several applications strive for CPU time and transmit bandwidth.

1 INTRODUCTION

Providing a deterministic service quality from a distributed application involves ensuring service quality in a whole chain of systems. Both the communication network and the computer platform on which the application is implemented have to provide service guarantees. In this paper we assume that the network provides the necessary service quality and focus on the mechanisms needed in the end-system computer platform to make it deliver the network

*This work is supported in part by the CEC DG III Esprit LTR project 21917 Pegasus II.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35580-1_16

J. D. Touch et al. (eds.), *Protocols for High-Speed Networks VI*

quality of service to the application. These mechanisms include providing service guarantees to the software implementing the communication protocols, i.e., guaranteed CPU time, and the allocation of transmit bandwidth on the network interface. In this paper we study CPU scheduling of a TCP/IP implementation, the scheduling of network interface transmit bandwidth and their interdependence in the context of the Nemesis operating system.

The Nemesis operating system [6] is designed to provide guaranteed quality of service (QoS) to applications. In order to provide guarantees it is necessary that all resources used by or on behalf of an application are accounted for correctly. In this respect, shared servers are a problem since they make it hard to charge the correct application for the resources used. In Nemesis the use of shared servers is instead reduced to a minimum. This leads to the *vertical structure* of Nemesis. Besides CPU time and disk I/O bandwidth, Nemesis regards transmit bandwidth on network interfaces as a resource that can be reserved.

We present a set of experiments which demonstrate the ability of Nemesis to provide the appropriate end-system communication guarantees for the application. First we show that the scheduling of transmit bandwidth can both be used as a rate limiter and to provide guaranteed transmit bandwidth. We measure the amount of CPU time an application needs in order to be able run the TCP/IP protocol stack and send data at a particular speed. Our experiments show that the CPU time to run the protocol stack increases linearly with the amount of data sent for a given packet size. We also show that the measured values hold even when several applications strive for CPU time and transmit bandwidth.

A possible usage of our scheme is an Internet service provider running one Nemesis host with several web servers for a number of customers with different performance requirements. Also other applications that need reliable data transfer can be run concurrently with guaranteed progress according to their reservations.

The contribution made in this paper is showing that the TCP/IP implementation in Nemesis can utilize the scheduling of CPU time and transmit bandwidth to provide the application with a guaranteed communication service. The scheduling of transmit bandwidth allows to rate-control TCP which can be useful for not exceeding a given traffic contract. We believe that this is difficult to achieve with traditional operating systems.

In the next section we briefly describe the features of Nemesis relevant for our work. In section three we give an overview on the design and implementation of the Nemesis TCP. Section four presents results and performance figures. In section five we discuss related work while section six concludes the paper.

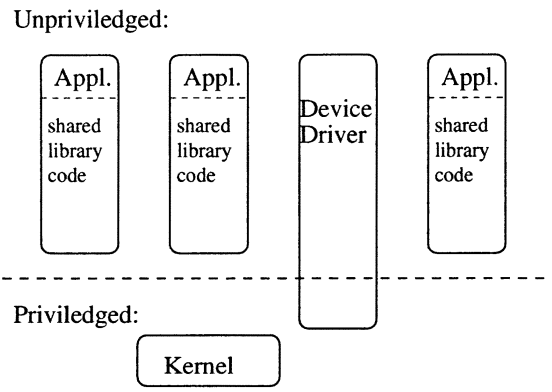

Figure 1 The Nemesis structure.

2 NEMESIS

The *vertical structure* of the Nemesis operating system is shown in Figure 1. Applications use shared library code to perform functionality usually associated with the operating system.

The Nemesis kernel is very small, consisting of the scheduler and activations of domains, the interrupt and trap handlers, support for inter-domain communication as well as some processor control. There are no kernel threads. A *domain* is similar to a Unix process. Trusted domains, such as device drivers, can register interrupt handlers and affect the processor mode. Despite having extra rights, trusted domains are scheduled like all other domains.

Nemesis has a single virtual address space which makes it easy to share data. Each domain still has its own memory protection on this address space, making lightweight inter-domain communication possible. Shared memory is used to transport the marshalled arguments and event counts synchronize the shared buffers.

Shared libraries export one or more strongly-typed *interfaces* which are specified in an interface description language called MIDDL. Interface descriptions comprise types, exceptions and methods. Having acquired a so-called binding or *interface reference* [6], domains can call methods of interfaces exported by other domains.

QoS Guarantees and Scheduling

Applications can request a share of the CPU time. Application requests consist of a tuple containing a period p, a time slice s and a flag x denoting

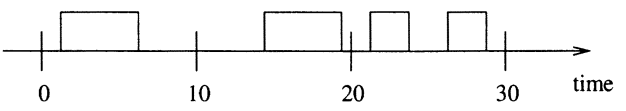

Figure 2 Scheduling of an application over three periods

whether the application wants to use *extra time*, i.e., a fair share of the non-allocated CPU time. If the reservation succeeds, the application is guaranteed a portion of s time units in each period of length p. The application is not guaranteed an atomic slice. The slice may be split up into several smaller parts.

An example is shown in Figure 2. An application has specified a period p of 10 time units, a slice time s of 5 units and the extra time flag x is set to `false`. In the first two periods the application receives one slice of 5 time units with the slice starting at different times. In the third period the application receives its slice time split up into two parts.

The scheduling is done using the Atropos scheduler [6] which internally uses an Earliest Deadline First algorithm. The deadlines are not specified by the applications but computed from the applications' specifications.

A similar scheme is used for the reservation of transmit bandwidth. Applications specify their reservations using the same tuple as described above, i.e., the reservations are expressed as transmission time and not as bandwidth. As for CPU time, the deadlines are not specified by the applications but computed from the applications' specifications. Setting the extra flag x to `true` means that the application wants a fair share of the non-allocated part of the transmit bandwidth.

Rbufs and I/0 Channels

Nemesis deploys a mechanism called *Rbufs* [3] for the inter-domain transport of bulk I/O through so-called I/O channels. Packets are formed using a data structure called I/O Record or *iorec*. An `iorec` consists of a header and a sequence of base pointer and length pairs similar to the `iovec` structure in Unix, with the header denoting how many pairs belong to the `iorec`. The base pointer points into a contiguous region of virtual address space called *Rbuf Data Area*, which is always backed by physical memory.

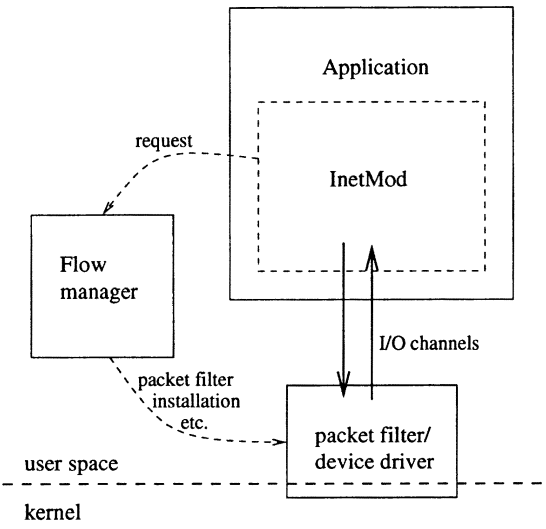

Figure 3 Architectural overview

3 TCP DESIGN AND IMPLEMENTATION

In this section we briefly describe the design and implementation of the Nemesis TCP, including some implementation problems that occurred due to the buffer handling scheme.

Overview

The Nemesis TCP implementation is partitioned between a shared library (*InetMod*) executed in the application domain and a trusted domain called the *flow manager*. The latter is responsible for both the synchronization of global resources, such as TCP port numbers, and for various control tasks like installing and removing packet filters. The main task of the packet filters is to demultiplex incoming packets to the right application at the lowest possible level. The packet filter is part of the device driver domain (see Figure 3).

Our implementation is based on the TCP/IP implementation of BSD [15]. As is done in BSD Unix, applications call TCP functions indirectly via a socket interface. For native Nemesis networking we provide two socket abstractions, namely `Socket` and `ServerSocket`.

As other TCP user-space implementations, such as Thekkat et. al. [13], our implementation uses threads, three for each connection. The *receiver thread* receives and processes incoming packets. Packets that should be delivered to

the application are queued in a FIFO queue (*to_app*) containing pointers to `iorecs`. The *timer thread*'s task is to trigger delayed acknowledgements, to force retransmissions, window and keepalive probes and to drop connections when the peer does not respond. The timer thread also prevents that the connection stays in the `FIN_WAIT_2` state forever and handles the deletion of the control block in the `TIME_WAIT` state.

The third thread is the application's thread. When the application wants to transmit data its thread also places the data into the send-queue and executes `tcp_output()` as well as `ip_output()` and the link-level output function. When the application asks to receive packets, the first packet(s) from the *to_app* queue are returned. If there are no packets, the thread blocks if so specified.

Buffer Handling

The buffer handling scheme in Nemesis is different from kernel-space TCP implementations. Applications need to supply the device drivers with empty buffers to receive data in. They must also reclaim buffers from the device driver when the corresponding packet has been sent.

Supplying the Device Driver with Empty Receive Buffers

On the arrival of a packet on a network interface, the packet filter determines the receiving application and the device driver copies the packet into receive buffers provided by the application. Thus, applications must *prime* the device driver, i.e., applications have to supply the device driver with empty receive buffers. In our TCP implementation, the receiver thread allocates receive buffers for header and payload of incoming packets and sends them to the device driver. The receive buffers are allocated from shared memory between the application and the device driver.

The application returns receive buffers to `InetMod` via an *extended* I/O channel. We call this I/O channel extended because it is an intra-domain extension of the I/O channel between the device driver and `InetMod` to the application program. `InetMod` then supplies the device driver with these receive buffers.

The number of empty receive buffers at the device driver limits the number of packets that a TCP connection can receive immediately. Thus, if the application or `InetMod` return the receive buffers slowly the device driver will run out of empty receive buffers and has to drop incoming packets. In our TCP implementation this situation is avoided by basing the calculation of the window advertisement on the number of empty receive buffers at the device driver

and the number of empty slots in the *to_app* queue between `InetMod` and the receiving application.

Reclaiming Used Transmit Buffers

After a packet has been transmitted, Nemesis applications must reclaim the corresponding transmit buffers from the network interface. When transmitting data, applications only allocate the memory for the payload. The memory for the header is allocated by `InetMod`. Thus, when applications reclaim used transmit buffers, `InetMod` only returns the payload buffer. Of course, `InetMod` cannot return a transmit buffer to the application before the corresponding packet has been acknowledged by the peer, in case the packet gets lost and has to be retransmitted. When an application reclaims transmit buffers and there are no transmit buffers available, the application's thread is blocked if so specified.

`InetMod` itself needs to reclaim the transmit buffers from the device driver. To achieve good performance it is important to do that at the right time. A naive solution is that `tcp_output()` reclaims the transmit buffers from the device driver after having called `ip_output()`. But this leads to bad performance since the device driver cannot return the transmit buffers before the packet has been put on the wire. Thus, it is advantageous to postpone this task until the device driver can return the transmit buffers without blocking.

Transmit Scheduling

The transmit scheduling is not part of `InetMod` but part of the device driver domain. It is implemented by a thread, the *TX thread.* The scheduled entities are the I/O channels to the device driver. The reservations are made by applications for each of their I/O channels by a method call.

The TX thread runs in an endless loop. It calls the Atropos scheduler to determine the next I/O channel to be served. The scheduler chooses an I/O channel based on the reservations. If this I/O channel has nothing to send, the scheduler searches for another suitable I/O channel. Thus, the call to the Atropos scheduler will return an I/O channel if there is an I/O channel that has pending packets and that has either a non-zero slice left in its current period or the extra flag set to `true`.

Thereafter the TX thread takes one packet out of the scheduled I/O channel, transmits the packet and charges the I/O channel for the transmission.

4 EXPERIMENTS

The experiments in the first part of this section deal with scheduling of interface transmission bandwidth only. In these experiments the applications have sufficient CPU time to produce data and process outgoing and incoming packets. Under heavier CPU load, however, solely reserving transmit bandwidth is not enough. If an application needs to send at a particular bandwidth, a sufficient part of the CPU time has to be reserved as well. Otherwise the application might not be able to produce data and run the protocol stacks fast enough. This is addressed in the experiments in the next part of this section that deals with the reservation of transmit bandwidth and CPU time. In the final part of this section we present performance results that show that the Nemesis TCP achieves good throughput.

Reserving Transmit Bandwidth

In this section we present two experiments. In the first experiment, we show how the choice of the period and slice time impacts TCP performance. The second experiment includes several transmitters that strive for transmit bandwidth. In this experiment we can also see the effect of the extra flag on the throughput that connections receive.

We have implemented a TCP sender application that sends MTU-sized packets as fast as possible to the peer. We have varied the QoS parameters for the transmit network bandwidth. The transmitter was a Pentium 200 running Nemesis with a 10 Mb/s 3c509 Ethernet card. The transmission of one MTU-sized packet (1500 bytes including TCP/IP headers) on a 10 Mb/s Ethernet takes about 1.2 ms. Setting the slice time to, e.g., twice this time allows the application to send two packets in every period.

One Transmitter

In our first experiment, the Nemesis box sends to a SPARCstation running SunOS. Figure 4 shows a section of a TCP connection with the slice time set to 2.4 ms (which allows to send up to two MTU-sized packets in one period)[1] and the period set to 50 ms. The extra time flag x is set to `false` in order to rate control TCP. The small lines with arrows represent the packets that are sent. The dotted line under the packets is the highest sequence number that has

[1] When we set the slice time to e.g. 3 ms (which allows to send "2.5" MTU-sized packets in every period), we will get periods with two mixed with periods with three packets sent. The value 2.4 ms is thus chosen to get a very regular traffic pattern.

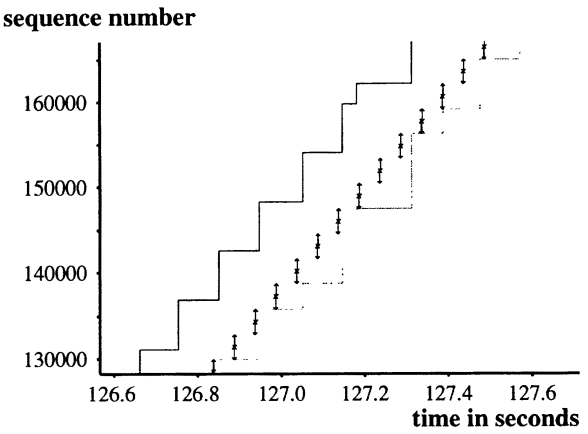

Figure 4 QoS TCP transmitting (tcpdump plot)

been acknowledged by the peer. The solid line above the packets is the window offered by the receiver. We can see that the sender transmits two packets, pauses almost 50 ms until the next two packets are sent and so on. Note that the transmission of packets is not triggered by the arrival of the acknowledgements but by the start of a new period.

From the period, slice time and the number of bytes to be sent per period we can compute the throughput for a connection. We have measured the throughput for a period of 200 ms and various slice times and compared the results with the theoretical values. For slice times up to 9.6 ms (maximum 8 MTU-sized packets per period) there are almost no discrepancies. However, for slice times that are larger than 9.6 ms, we do get very varying results. The explanation is that the receiver's window fills when the transmitter sends many packets during some periods and then the window offered by the peer varies depending on how fast the receiver acknowledges data. This prevents the transmitter from utilizing the guaranteed transmit bandwidth. However, the aim of rate-control is to set upper limits.

When transmitting over long distances or low-bandwidth connections, TCP flow control determines the overall traffic pattern. However, when the peer acknowledges several TCP segments, the Nemesis TCP does not reply by transmitting immediately a bulk of data until the receiver's window is filled again, but spreads out the packets depending on the chosen period and slice time.

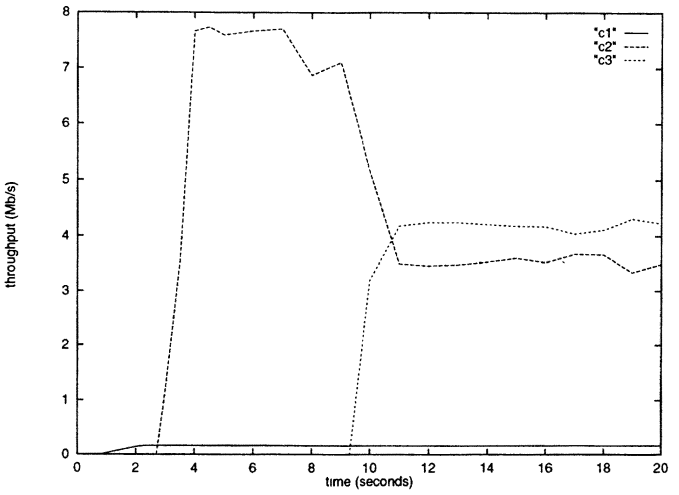

Figure 5 Time-throughput graph for 3 connections

Several Transmitters

In our second experiment, the Nemesis machine sends to a SPARCstation called *kay* and another SPARCstation (called *garuda*) running NetBSD. *Garuda* is equipped with a 10 Mb/s Ethernet card and sits on the same LAN as the Nemesis machine. The result is shown in Figure 5. First we start a TCP connection (*c1*) to *kay* with a period of 200 ms, a slice time of 3.6 ms and the extra flag set to `false`. This connection consistently receives the same bandwidth (about 0.17 Mb/s) which conforms to its guarantees.

About three seconds later, we start a second TCP transmitter (*c2*) to *garuda* with a period of 10 ms, a slice time of 1.2 ms and the extra time flag set to `true`. This means that *c2* wants to make use of the non-allocated part of the transmit bandwidth but also has some guaranteed transmit bandwidth. Since *c1* only consumes a small part of the available bandwidth, *c2* initially receives a throughput of about 7.5 Mb/s, with some variation due to how fast the receiver acknowledges the data. Some seconds later the third connection (*c3*), also to *garuda*, is started. It specifies a period of 10 ms, a slice time of 6 ms and the extra flag is set to `false`. The large slice time is chosen to reserve a large fraction of the available transmit bandwidth. As soon as *c3* has started, the non-allocated part of the transmit bandwidth decreases, and *c2* therefore receives much less transmit bandwidth (about 3.5 Mb/s).

Unfortunately, *c3* does not receive the guaranteed bandwidth of 5.6 Mb/s

but only 4.2 Mb/s. The reason for this lower than expected throughput is that, quite often, the sender is not allowed to transmit since the receiver's window is closed. This happens because the acknowledgements that the receiver sends are often delayed and arrive in bursts instead of arriving regularly (we saw a similar behavior when we ran Linux on both the sender and a receiver on the same LAN and transmitted as fast as possible). The delay is most likely caused by contention for the Ethernet media. We use a 10 Mb/s hub and have two fast connections. Under these conditions the shared Ethernet becomes a bottleneck. On the other hand, since *c2* has the extra flag set, the overall throughput does not decrease because *c2* makes use of the bandwidth that *c3* cannot utilize due to the closed window.

Reserving Transmit Bandwidth and CPU Time

In this section we present experiments that show the impact of CPU reservations on the achieved bandwidth of TCP applications. All connections are on a LAN with both transmitter and receiver running de4x5 100 Mb/s Ethernet cards. The transmitter is a Nemesis machine while the receiver runs Linux. In the following experiments we run a TCP application called *tcp_send* that transmits MTU-sized packets as fast as possible. We also run an application called *carnage*. This application has the property that it makes use of all its reservations. If it is guaranteed a certain fraction of the CPU time, it will make use of this fraction.

Impact of CPU Time on Achieved Throughput

In this experiment *tcp_send* reserves a large fraction of the network transmit bandwidth and has the extra time flag *x* set to `true`. Since *tcp_send* is the only transmitting application it will receive almost all the available transmit bandwidth. The CPU reservation request of *tcp_send* specifies a period of 10 ms, a slice time of 2 ms and the extra time flag is set to `true`. Thus *tcp_send* will receive a fraction of about 20 % of the CPU time and a fair share of the CPU time not used by other applications. *carnage* has specified a period of 20 ms, the extra flag is set to `false` and we vary the slice time to control the fraction of the CPU time that *carnage* receives. The more CPU time *carnage* receives the less CPU time is left for *tcp_send*. *tcp_send* will receive its slice time every period but the non-allocated part of the CPU time will decrease when *carnage's* slice time increases.

Figure 6 shows how the decreasing fraction of the CPU impacts the throughput that *tcp_send* achieves. Since *tcp_send* reserves 20 % of the CPU time, we

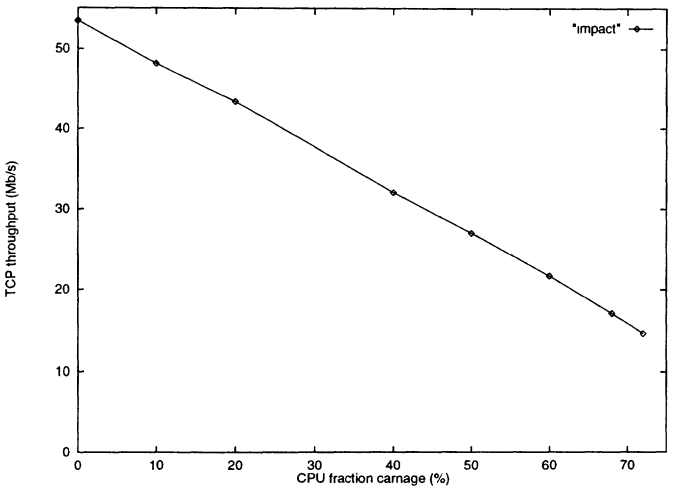

Figure 6 Impact of *carnage* application on TCP bandwidth

set the maximum reservation for *carnage* to 72 %. We do not make a reservation of 80 % for *carnage* since there are also other domains running such as the flow manager and several device drivers. In particular the Ethernet driver consumes a non-negligible part of the CPU time in the experiments in this section.

When only *tcp_send* is running, it achieves a throughput of about 53 Mb/s. The figure shows that the throughput decreases proportionally to the fraction of CPU time used by *carnage*.

Reserving CPU Time to Achieve Throughput

In this experiment we assume that we have an application that wants to transmit data with a desired throughput. The goal is to determine the fraction of the CPU time an application needs in order to produce data and run the protocol stack fast enough to be able to transmit the desired amount of data. This can be important when you have reserved network bandwidth and you want to make sure that your application really achieves the bandwidth you pay for but you still want to have your text editor and other applications running.

The application *tcp_send* reserves the desired transmit bandwidth on the network interface. When the desired throughput is e.g. 10 Mb/s, *tcp_send* specifies a period of 1 ms, a slice time of 110 μs and sets the extra flag to `false`. The transmission of one MTU-sized packet (1500 bytes including TCP/IP headers) takes about 120 μs on a 100 Mb/s Ethernet. Since we need to send slightly less than one packet per millisecond to achieve a throughput of

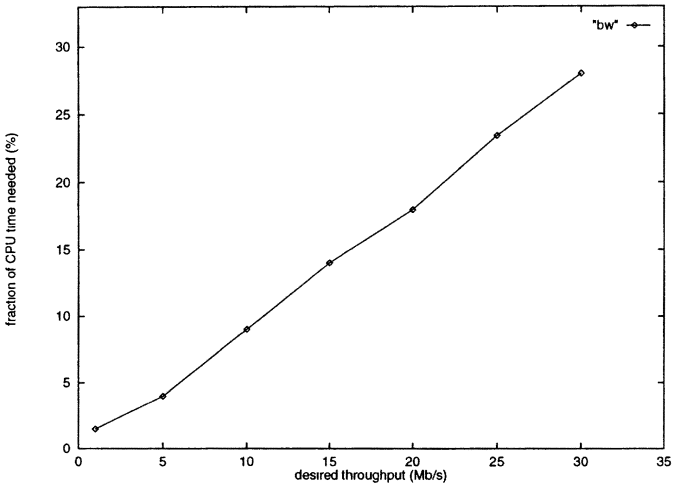

Figure 7 Fraction of CPU necessary to send particular amount of data

10 Mb/s, setting the slice time to 110 μs seems to be reasonable. This has been confirmed by measurements.

In Figure 7 we show the fraction of the CPU time the application *tcp_send* needs to reserve to be able to transmit the desired amount of data. Since *tcp_send* does not do any data processing and mainly sends and retrieves buffers this can be seen as a minimum CPU fraction necessary to run the TCP protocol stack. However, the exact fraction of the CPU time needed depends on the packet length, the CPU, the network adapter and other system components.

Figure 7 shows that the CPU fraction needed increases linearly with the amount of data sent.

Several Applications Reserving Transmit Bandwidth and CPU Time

In the next experiment we run three *tcp_send* applications (*s1*, *s2* and *s3*). *s1* and *s2* both want to transmit data at a speed of 10 Mb/s and *s3* at a speed of 5 Mb/s. The applications set the corresponding QoS-parameters (see Table 1) using the tuple (period, slice time, extra flag).

The fractions of the CPU share are the ones we determined in the previous experiment (see Figure 7), i.e., 10 % for *s1* and *s2* (*s1* and *s2* have different period and slice times but reserve the same fraction of CPU time and transmit bandwidth) and 5 % for *s3*. We also run a *carnage* application that has the extra flag set to `true` and thus consumes all the non-allocated CPU time.

Table 1 also shows that all applications receive their desired throughput.

Table 1 Experiment with three transmitting applications

app.	aim	TX QoS	CPU QoS	Throughput
s1	10 Mb/s	1 ms, 110 μs, F	10 ms, 1 ms, F	10.1 Mb/s
s2	10 Mb/s	2 ms, 220 μs, F	20 ms, 2 ms, F	10.1 Mb/s
s3	5 Mb/s	2 ms, 110 μs, F	20 ms, 1 ms, F	5.1 Mb/s

Performance

Nemesis and its network architecture are primarily designed to support QoS. Raw throughput performance is secondary. We nevertheless think that the throughput we achieved with the TCP is reasonable. We have run *tcp_send* between a Pentium 200 Nemesis machine and another Pentium 200 machine running Linux. Both machines are equipped with 100 Mb/s de4x5 Ethernet cards. We received a throughput of more than 58 Mb/s. This is lower than the 81 Mb/s we received when running Linux on both machines but we still think this is reasonable considering the primary goal with Nemesis. We believe that it is possible to optimize the Nemesis TCP a lot more. A problem in our architecture is that the transmitting application and the device driver run as different domains and are thus scheduled separately. The deployment of a *path* mechanism as in Scout [10] would probably increase throughput performance.

5 RELATED WORK

Satyavolu *et. al.* [12] have investigated in methods for controlling the rate of TCP applications. Our work deals with the end system, whereas their techniques can be applied in ATM edge devices and Internet routers. Crowcroft and Oechslin [4] do not rate control TCP traffic but limit the throughput of TCP connections by decreasing the size of the receive buffer in order to achieve weighted proportional fairness. Black *et. al.* [2] have used Nemesis and the Atropos scheduler for UDP. In contrast to TCP, UDP does not have schemes for, e.g., congestion and flow control which can prevent a transmitter from sending although there are packets ready for transmission. Stride scheduling [14] has been used for both CPU and device driver scheduling but the authors do not report on any results of an integrated use.

Other architectures than Nemesis for providing QoS in an end system have been proposed. Yau and Lam [16] propose an end system architecture that supports networking with quality of service guarantees. They also use scheduling for both CPU and network interface access by deploying an adaptive rate-

controlled (ARC) scheduler. The AQUA framework of Lakshman and Yavatkar [8] is used to manage CPU and network I/O resources in an integrated fashion. Their scheme is adaptive while we use reservations to provide QoS. Both Yau's and Lakshman's architectures have been implemented in Solaris whereas we use the Nemesis operating system which is designed from scratch to support quality of service. Gopalakrishnan and Parulkar [5] have implemented an efficient user-space protocol that supports QoS using real-time upcalls (RTU's). They achieve impressive performance running TCP/IP over ATM. Their mechanisms also allow to minimize delay or maximize throughput. The architecture the authors present in their paper deals with protocol processing only. In our scheme, the binding between an application and the underlying protocol processing unit is more explicit because protocol processing is performed by the application itself using shared libraries. Rajkumar et. al. [9] aim for predictable communication protocol processing in Real-Time Mach by using *processor reserves*. As in Nemesis, library code in the application implements protocol processing. In contrast to Nemesis, their scheme relies on processor reservations only. Thus, it is possible that an application that is able to produce data faster can achieve higher throughput than an application that has higher processor reserves but needs more time to produce the data that it transmits. Another difference is that processor reserves are made per thread and the real-time socket library comprises at least four threads.

The Rialto operating system [7] supports coexisting independent real-time and non-real-time programs by sharing the limited physical resources available to them. In one of their experiments, the authors study the influence of CPU reservations over video rendering fidelity. Due to the interdependency between frames there is no linear relationship between the CPU reservation and the frames not rendered.

A different way of providing quality of service in an end-system are feedback-based approaches such as G. Beaton's [1]. One of the goals of this approach is to avoid the complexity of other architectures like Nahrstedt's and Smith's QoS Broker [11].

6 CONCLUSIONS

We have designed and implemented a TCP for Nemesis, a vertically integrated operating system that can guarantee resources such as CPU time, disk I/O bandwidth and transmit network bandwidth to applications. Our experiments show that the Nemesis transmit scheduler rate-controls TCP traffic when applications choose not to utilize non-allocated bandwidth. Applications that

want to make use of the non-allocated transmit bandwidth receive the guaranteed bandwidth plus a share of the non-allocated bandwidth. We show that the CPU time needed to run the protocol stack increases linearly with the amount of data sent for a given packet size. When networked applications reserve enough CPU time and transmit bandwidth they receive sufficient resources even when several applications strive for both CPU time and transmit bandwidth.

Thus, Nemesis allows us to run several applications, which require both CPU time and transmit bandwidth, concurrently, making sure each of them receives the guaranteed resources. Possible applications include concurrently running web servers with different performance requirements as well as other applications requiring reliable data transfer.

7 FUTURE WORK

After having optimized the Nemesis TCP we plan to investigate how Nemesis can be extended to perform as an end-host in a differentiated services platform. We believe that we might benefit from the Nemesis architecture here. An obvious advantage is that applications that have reserved enough resources can produce their data in time, independent of other applications.

8 ACKNOWLEDGEMENTS

Many people have contributed to Nemesis during the last years. Without their work we could not have written this paper. In particular Austin Donnelly's work has been of importance to us.

The authors would also like to thank Per Gunningberg for valuable comments on an earlier draft of this paper.

References

[1] Gordon Beaton. A feedback-based quality of service management scheme. In *Third International Workshop on High Performance Protocol Architectures (HIPPARCH '97)*, Uppsala, Sweden, June 12–13 1997.

[2] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *IEEE 22nd Annual Conference on Computer Networks (LCN)*, pages 179–188, November 2–5 1997.

[3] R.J. Black. Explicit network scheduling. Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation.

[4] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, July 1998.

[5] R. Gopalakrishnan and G. M. Parulkar. Efficient user space protocol implementations with qos guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4):374–388, August 1998.

[6] I.M.Leslie, D.McAuley, R.Black, T.Roscoe, P.Barham, D.Evers, R.Fairbanks, and E.Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[7] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, and M. Rosu. An overview of the rialto real-time architecture. In *Seventh ACM SIGOPS European Workshop*, pages 249–256, Connemara, Ireland, September 1996.

[8] K. Lakshman, R. Yavatkar, and R. Finkel. Integrated cpu and network-i/o qos management in an endsystem. In *Int. Workshop on Quality of Service (IWQoS)*, pages 167–178, 1997.

[9] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *Proceedings of the Real-Time Technology and Application Symposium*, pages 115–123, June 1996.

[10] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Symposium on Operating Systems Design and Implementation*, pages 153–167, October 1996.

[11] K. Nahrstedt and J. Smith. The qos broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[12] Ramakrishna Satyavolu, Ketan Duvedi, and Shivkumar Kalyanaraman. Explicit rate control of tcp applications. ATM Forum Document, February 1999. ATM Forum Document Number: ATM_Forum/98-0152R1.

[13] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[14] C.A. Waldspurger and W.E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical report, MIT Laboratory for Computer Science, 1995.

[15] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, 1995. ISBN 0-201-63354-X.

[16] David K.Y. Yau and Simon S. Lam. Migrating sockets - end system support for networking with quality of service guarantees. *IEEE/ACM Transactions on Networking*, 6(6):700–716, December 1998.