

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Exploring TCP Stream Rate Control in LAN Environments

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Nirdosh J. Shah

June 1999

Thesis Committee:

Professor Mart Molle, Chairperson

Professor Satish K. Tripathi

Professor Thomas H. Payne

Copyright by
Nirdosh J. Shah
1999

The Thesis of Nirdosh J. Shah is approved:

Committee Chairperson

University of California, Riverside

Acknowledgements

Although the title page lists only my name, this document has been affected in many significant ways by the wonderful people around me. First and foremost is my wife, Heidi, who gave up so many evenings, weekends, and holidays without complaint. Without her support from the home front, this thesis would not have been possible.

A big hug to my parents, Jagdish and Manjari Shah, who have always been a constant source of support and encouragement. Their infinite patience has been nothing short of remarkable. Everything I am is because of them. Another hug to the rest of my family, Mona, Paresh, and Sangeet, who have provided support and periodic sanity checks without fail.

A special thank you to my advisor, Mart Molle, who has provided an incredible amount of help over the last two years. This work could never have been completed without his insight. Thanks also to my committee, Dr. Thomas Payne and Dean Satish Tripathi for taking time out of their busy schedules.

Many thanks for great technical advise and insight to: Vishnu Priya, Alexey Kruztnov, Dan Mick, Alan Cox, and Sally Floyd.

ABSTRACT OF THE THESIS

Exploring TCP Stream Rate Control in LAN Environments

by

Nirdosh J. Shah

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 1999
Professor Mart Molle, Chairperson

The demand is rising for multimedia applications requiring high bandwidth, low latency networks, such as video conferencing. Most local area networks, however, are not ready to cope with the additional load and thus the service quality will not meet user expectations. Mechanisms to control bandwidth such that applications can be guaranteed a certain level of network performance are either out of reach because of cost (e.g. ATM), only work in WAN environments (e.g. QoS-able routers), or have a limited scope (e.g. Linux's output-only QoS support).

For this thesis we explore a solution to this problem by modifying the Linux kernel such that it can control input streams using mechanisms already found in TCP. We believe that in combination with its existing output-only QoS support, a complete solution can be presented.

Our results are presented in two steps: the first validates our assertion that local area networks are in fact overly congested and require mechanisms to control QoS. The second shows that we are able to control bandwidth using our modified algorithm to compute TCP window sizes. Our solution does not require additional timers or the generation of additional control packets. However, the method has a limited range of rate adjustments and is only effective if the target rate is above some system dependent lower bound.

Contents

1	Introduction	1
1.1	Emergence of Real Time Multimedia Applications	1
1.2	The Need for Bandwidth Control	5
1.2.1	ATM vs. Ethernet	6
1.2.2	Where the Control is Needed	7
1.2.3	A Real Life Example	7
1.3	Existing Mechanisms for Bandwidth Allocation and Control . .	8
1.3.1	Addressing the WAN	9
1.3.2	Special Algorithms	10
1.3.3	The Standards That Are Works in Progress	10
1.3.4	Wide-Scale Adoption	11
1.3.5	Throwing Bandwidth at the Problem	12
1.4	A Possible Solution	13
1.4.1	Why not a simulation?	14
1.4.2	About Linux	15

2	Network Traffic Environment	16
2.1	The CE-CERT Network Environment	19
2.2	Summary and Comparison of Findings	20
2.2.1	General Statistics	21
2.2.2	Network Protocols	24
2.3	Analysis	27
2.3.1	The Migration to Windows	28
2.3.2	Faster Hosts	30
2.3.3	Conclusion	30
3	Quality of Service Algorithms	32
3.1	CBQ - Class Based Queueing	32
3.1.1	Requirements for Hierarchical Link Sharing	33
3.1.2	CBQ Terminology	35
3.1.3	CBQ's Rules of Behavior	37
3.1.4	CBQ in Linux	40
3.2	The Resource reSerVation Protocol (RSVP)	41
3.2.1	How does RSVP fit into the system?	42
3.2.2	How Does RSVP Work?	44
3.2.3	Key Attributes of RSVP [9]	45
3.2.4	RSVP in Linux	47
4	A Tour of Linux Network Code	48
4.1	Network Code Organization	51

4.1.1	Device Drivers	52
4.1.2	The Data Link Layer	55
4.1.3	The Network Layer	58
4.1.4	The Transport Layer	64
4.2	Congestion Control	67
4.2.1	Delayed Acks	67
4.2.2	Nagle's Algorithm	69
4.2.3	TCP Sliding Window	72
4.3	Quality of Service in Linux	75
4.3.1	Where does QoS Fit?	75
4.3.2	Separation of the Classifiers, Schedulers, and Queues	77
4.3.3	Linux Specific Details	79
5	Implementing Rate Control	82
5.1	Keeping Track of Rates	82
5.1.1	“Chunkifying”	83
5.1.2	Computing the Rates	85
5.1.3	Aging Rates	86
5.2	Controlling the Rate	86
5.2.1	Handling TCP	87
5.2.2	Handling UDP	88
5.3	The /proc Interface	90
5.4	Interfacing with CBQ	92

5.4.1	A Usage Example	94
6	Experimental Results	95
6.1	Experimental Network Configuration	95
6.2	<code>ttcp</code> Over Ethernet	97
6.2.1	Controlling <code>ttcp</code> to a Fixed Transfer Rate	98
6.3	Sensitivity of Throughput to Window Size	101
6.3.1	Findings	106
6.4	Evaluating the MSS Changeability	107
6.5	Connections with Differing MSS Values	108
6.5.1	Findings	109
6.6	Changing the Window Sizing Algorithm	111
6.7	<code>ttcp</code> over a T1 Link	112
6.7.1	Findings	113
6.8	<code>ttcp</code> Over A 38.8kbps Link	114
6.8.1	Findings	116
6.9	Multiple Concurrent <code>ttcp</code> 's Over Ethernet	117
6.9.1	Findings	118
6.10	Multiple Concurrent <code>ttcp</code> over a T1 Link	119
6.10.1	Findings	120
6.11	CBQ vs. Our Algorithm	120
6.11.1	Findings	122
7	Conclusions	124

List of Figures

2.1	CE-CERT's Network Topology	21
2.2	Bandwidth Consumption Over Time	23
2.3	Packet Interarrival Time	26
3.1	CBQ's Hierarchical Design	38
3.2	CBQ Limits	39
3.3	RSVP Design	43
4.1	OSI 7-layer model	52
4.2	Linux 4-layer model	53
4.3	How TCP/IP maps into the OSI model	54
4.4	Relationship between actual Linux functions and the OSI model.	55
4.5	IP Header	61
4.6	TCP Header	65
4.7	UDP Header	66
4.8	TCP Sliding Window at the Receiver	73
4.9	Where QoS fits in the network stack	76

4.10	The relationship between classifiers, schedulers, and queues . . .	77
6.1	<code>ttcp</code> Experimental Result: Bytes Per Second	99
6.2	<code>ttcp</code> Experimental Result: TCP Window Size vs Time	100
6.3	<code>ttcp</code> Window Size vs. Bytes/Sec	101
6.4	Bytes/Second for Varying Window Sizes (32K to 1K)	103
6.5	Bytes/Second for Varying Window Sizes (2K to 1K)	105
6.6	Bytes/Second for Varying MSS Sizes	110
6.7	<code>ttcp</code> Rates Over a T1	114
6.8	Average speed of 4 <code>ttcp</code> runs over a T1	115
6.9	<code>ttcp</code> Rates versus Time in Seconds over a 38.8kbps PPP Link	117
6.10	A single stream's rate out of five concurrent streams at 60% . .	119
6.11	Average rate of a stream versus number of concurrent streams	120
6.12	Multiple <code>ttcp</code> runs over a T1	121
6.13	Performance when using CBQ to control incoming streams . .	123

Chapter 1

Introduction

1.1 Emergence of Real Time Multimedia Applications

With the Internet quickly becoming ubiquitous in our culture, its growth has been nothing short of phenomenal. Availability of commercial services has grown to the point that it is almost entirely possible to live a normal day to day life without ever leaving your web browser. To quote a peer in the information technology industry, "I bought groceries over the web today."

The growth rate of the Internet is astounding. As of this writing, it is estimated that over 30% of American homes have Internet access. Projected growth rates indicate numbers as high as 50% within the next 2-3 years. Internet access for employees within the commercial sector is seeing similar

surges as more companies are seeing Internet e-mail as part of their basic work model. A strong web presence is quickly becoming part of the basic model as well, especially given the rate at which information disseminates over the Internet compared to traditional media.

In order to support this level of Internet activity, maintaining network infrastructure has become a major issue. The number of Internet Service Providers (ISP's) has risen to the point that we can consider connectivity to be available almost anywhere. Given this trend, commercial entities have begun exploring the possibilities of closing the gap between remote offices through the use of Internet based video conferencing, virtual whiteboards, and other such tools. This level of remote collaboration has come at the expense of greatly increased bandwidth requirements. MCI/WorldCom, for instance, expanded their Internet backbone capacity by a factor of 4 in 1997 alone [17].

The demands placed on Local Area Networks have increased substantially as well. Operating systems, to be of any significant use in an office environment, now rely on network services. Multimedia protocols that support video conferencing, IP telephony, and similar technologies require an additional significant slice of bandwidth.

With all of the demands on the network, it is inevitable that contention

for available bandwidth is becoming a significant issue. Network congestion occurs because it is simply not feasible to service everyone's request at the same time. While this may be acceptable for many applications (e.g., packets containing e-mail), it is not acceptable for streams of data that must be presented to the receiver in real time. (e.g., video)

Since network congestion can potentially interrupt real time data streams, mechanisms to guarantee Quality of Service (QoS) for those applications that require it need to be put into place. There are two approaches to this: using the Resource reSerVation Protocol (RSVP) [6] and Priority Tagging [10]. These technologies are currently being discussed amongst Internet Engineering Task Force (IETF) groups as part of an effort to establish standards.

The underlying principle in RSVP is never to commit to the delivery of QoS sensitive traffic unless it knows in advance the network will be able to honor its delivery requirements. For each person who wishes to establish a network connection with a guaranteed amount of bandwidth, a request is made from their host to the remote host. Each router along the path verifies that the bandwidth is available and that the requestor has the necessary permissions. If the requestor does not have the necessary permissions, the router rejects the connection. Once such a connection is established, the

path between the two hosts remains static and traffic between them receives the guaranteed amount of bandwidth regardless of other traffic sharing the link [6].

Priority Tagging is fundamentally different than RSVP. Instead of establishing a static path with guaranteed bandwidth, each packet from the sender is labelled with a special value to indicate its importance. Routers in the network can then examine the priority label in each packet and reorder their queues to allow higher priority packets to go first [10].

The philosophical difference between the two schemes is relatively straightforward. RSVP guarantees that priority connections will never oversubscribe a particular link. This comes at the risk of rejecting some high priority requests and allowing the allotments to some users to be under-utilized. Priority Tagging doesn't have the problem of under-utilization, but it does have the risk of being over-subscribed. Should a large number of high priority packets arrive at about the same time, none will get rejected because the available bandwidth is inadequate, but they may suffer impaired quality of service due to the congestion.

While the discussion continues between the two differing viewpoints, some networking vendors are releasing products based on preliminary reports. This can be bad for the consumer since inter-vendor inter-operability cannot be

guaranteed. Furthermore, it is rare that an Internet Service Provider (ISP) would offer support for protocols that are still in flux. Thus, most network hardware consumers will end up having to wait for the standards to be fully settled on and products conforming to these standards to be released.

1.2 The Need for Bandwidth Control

Before we can evaluate our options for bandwidth control, we need to stop and understand the two environments they need to work in: LANs and WANs. In the LAN environment, there is more available bandwidth. However, it must in turn be shared with more resources as well (e.g. file sharing, printing, etc.) The LAN environment is also less complex than WAN environments – most LAN traffic tends to be localized and thus does not encounter routers, bridges, or gateways. LANs tend to have low latency due to the short distances involved. On the other hand, LANs based around Ethernet may also exhibit some peculiar performance characteristics because of CSMA/CD [18]. In contrast, characteristics exhibited by WANs include higher latencies between routers, lower available bandwidths, and more complex packet forwarding rules. All of these are accepted trade offs for the benefit of distance.

1.2.1 ATM vs. Ethernet

Although Ethernet is by far the most common local area network technology, it is not the only one. Asynchronous Transfer Mode (ATM) is another well known option.[20] ATM's design philosophy is that at the expense of additional, more complicated (and thus costly) hardware, it is possible to provide well controlled bandwidth. Furthermore, ATM's designers foresaw the desire to unify voice and data networks and thus provided means to support both.

Because of the increased funding required to purchase a network with ATM going down to the desktop, it has not taken a strong foothold in the LAN market. ATM has had more success in WANs and network cores, two places where fine grained control and speed are critical issues, both of which ATM provide.

Ethernet on the other hand is based on the philosophy of designing it simple and keeping it low cost. By keeping the technology at a lower cost, network congestion caused by lack of bandwidth controls can be remedied by purchasing more bandwidth.

1.2.2 Where the Control is Needed

With ATM technology primarily in network cores and wide area network solutions, Ethernet remains the primary network found connecting desktop systems. By its nature, Ethernet lends itself to a network topology where desktop hosts connect with lower speed connections to workgroups and higher speed switches connect workgroups with one another. With gigabit Ethernet becoming more readily available, gigabit Ethernet is finding its way into network cores to tie key servers and high speed switches together.

With the topology of a well designed Ethernet network being as we just described it, we see two key points at which network congestion needs to be addressed:

1. WAN connections.
2. The network's edges: host to workgroup hubs/switches.

1.2.3 A Real Life Example

Updating and installing software for a large user base usually requires that some sort of automated system exist. These systems work by using a server to “push” a copy of the software to the client over the network, transparent to the user.

In a situation where the user is doing something that is bandwidth critical, e.g. video conferencing, adding a remote update at the same time will create contention on the link. This fight for bandwidth will result in poor image quality and choppy sound. Obviously this is not acceptable to the user.

What is needed in this situation is some method of allowing the client to control the rate at which the server transmits the stream. Because the client is aware that a bandwidth critical stream is on-going, the server's stream can be slowed down such that the image and sound quality are not affected.

As of this writing, software which can perform this function in a non-proprietary way does not exist.

1.3 Existing Mechanisms for Bandwidth Allocation and Control

Anyone reading trade magazines will see many advertisements and stories on mechanisms for controlling bandwidth [15]. has an impressive laundry list of companies offering solutions. In most cases these solutions are for Wide Area Network connections such as links to the Internet; software solutions that can exist on a company LAN must reside at the gateway level.

In this section, we address why existing solutions do not meet our criteria for addressing bandwidth at the host level as well as other miscellaneous concerns.

1.3.1 Addressing the WAN

In products that address bandwidth control issues with wide area networks (WAN), the basic assumption regarding traffic differs from that of traffic on a LAN. WAN connections tend to have lower bandwidth with significantly higher latencies. The cost of an error in the control mechanism is also higher since WAN connections are typically leased on a month to month basis. Bandwidth lost to inappropriate usage is money lost.

As a result, systems that address WANs are often able to support more complex solutions since the time between arriving packets is substantially lower than say a 100Mbit Ethernet connection. In the situation of addressing a congestion problem at the host level, it must be remembered that the host's primary job is to perform user's tasks and thus any additional bandwidth control must work at a very low computational cost.

1.3.2 Special Algorithms

If a product requires that both sides of a connection utilize a specific protocol, whether it is a standard in development or a proprietary solution, it poses a limitation in deployment. You must either buy two identical units or in the case of connecting with an ISP, find an ISP which supports the special protocol.

The fundamental problem with proprietary solutions is that they are proprietary. Unless support continues to come from the company that originally developed it, the user is left without support. Furthermore, when the goal is to be able to connect with other users, sticking with proprietary solutions may lead to trouble in the future.

1.3.3 The Standards That Are Works in Progress

IEEE and IETF standards groups are working diligently to come to agreements on QoS algorithms. However, until the standards are finalized, there is room for change. This is a common problem with manufacturers trying to get to market as quickly as possible with standards compliant hardware and software. When standards change, their products need upgrades to become compliant again and it is simply impossible to guarantee the upgraded version won't cause an existing configuration to break. For sites that deal with

multiple vendors, this can be especially troublesome since not all vendors will be able to bring their products up to specification at the same time.

1.3.4 Wide-Scale Adoption

A common bandwidth control mechanism being discussed is the Resource reSerVation Protocol (RSVP). One of the core necessities of RSVP is that everyone using the system agree on a standard means of authenticating individuals and charging back for bandwidth requested. This sort of solution has the possibility of working in a private corporate network. However, even that can be tricky unless their IS staff has a global mechanism for authenticating users and their rights through a directory service or similar item.

Given the complexity of providing a centralized authentication method within a single organization, anyone who believes that RSVP's global authentication is going to be the future of the Internet is woefully optimistic. The RFC for RSVP itself cites the problem of global adoption has being problematic and as of this writing has not yet proposed a solution to it[6].

1.3.5 Throwing Bandwidth at the Problem

A common solution to bandwidth issues is accomplished through a more brute force technique: Buying additional routers and switches to segment congested areas of the LAN.

Although this technique is preferred by network administrators everywhere, the author feels authoritative in citing his own experiences as a systems/network administrator for both commercial and academic organizations in saying that spending money to upgrade a network happens much less often than we'd like to believe. Numerous war stories on how much teeth pulling it takes for some organizations to put a minimal amount of money into network infrastructure can be heard in newsgroups and conferences. “;Login”, the journal published by SAGE (a subset of USENIX) goes as far as publishing a series of articles on dealing with these issues.

Assuming one does have necessary funds and can purchase the appropriate gear, a multitude of choices exists. Routers, switches, routing switches and bridging hubs are all available to help reduce the size of one's collision domains thus reducing the utilization on each segment and improving each user's performance.

There is a catch, of course. Unless a significant amount of money is spent on putting all of the users on a switch of some kind, there will be users who

will have to share bandwidth and thus it still takes only one user doing something bandwidth intensive to make real time multimedia applications tricky at best for others on the network.

Unfortunately, no matter how much bandwidth is thrown at the problem, there will always be a bottleneck somewhere. Whether it is older hardware which cannot be upgraded to support the available bandwidth or legacy applications which is no longer supported and thus unable to be converted to support available QoS mechanisms, throwing bandwidth at the problem is not the ideal solution.

1.4 A Possible Solution

So far we have discussed what our networks are composed of, what their topologies look like, and what solutions exist to provide quality of service to our users. What we found was that in the two places which need help on controlling bandwidth utilization, WAN connections and links to hosts, only WAN connections have been truly addressed.

In this thesis, I show that congestion can occur at the host level; that is the traffic on the wire between a host and the hub/switch port can limit the quality of service for network streams which need constant performance.

The limitation is often caused by the capture effect from other best effort streams such as web, ftp, and network disk services.

My proposal is to examine whether modifying the Linux kernel such that it can use mechanisms built into TCP (sliding window control) to control send rates to a host, thus giving network streams which need additional bandwidth the necessary time on the wire. This proposal does not require other hosts to use this system or any other QoS mechanism for that matter.

If the improvements in Quality of Service can be accomplished with my proposal, I hope to contribute the findings back into the Linux source tree for public distribution.

1.4.1 Why not a simulation?

While a well-written simulation can prove or disprove an idea, it is best used for situations where actually implementing the project would require resources that are simply out of reach for a small scale project.

With this project, I have the unique opportunity to implement my proposal in a real, well regarded and commercially used operating system: Linux. I felt Linux as a platform would allow me to perform more real to life experiments since special "for simulation use-only" applications would not have to

be written. Furthermore, network workload models, which has been traditionally difficult to create would not be needed. Instead, my “thesis-version” of Linux would just have to be plugged into an existing network and put into action using live applications.

1.4.2 About Linux

Linux is an open source operating system which has been in development since 1991. It is a complete UNIX-like kernel with full networking capability and is available in many different distributions with compilers, editors, various servers, and clients. Within the last year, Linux has received a great deal of press and as a result is now considered a main stream operating system. Many commercial organizations are deploying Linux as server platforms and are in general quite happy with its performance and stability.

Chapter 2

Network Traffic Environment

For the purposes of taking a closer look at local area network load, we examine the most common type of LAN environment: Ethernet, developed at Xerox Palo Alto Research Center (PARC) in the 1970's [4]. The success of Ethernet has been attributed to its simplicity, low cost, reasonably large bandwidth, and longevity. Bob Metcalf, the father of Ethernet, estimated that over 100 million Ethernet cards had been sold as of the early 1990's.

The popularity of Ethernet has led to repeated studies of its performance characteristics. An early, but significant paper by Shoch and Hupp [23] from Xerox PARC suggested, and it is now commonly accepted [14], that Ethernet under normal workloads is lightly loaded.

In 1990, Riccardo Gusella reexamined this area of research [11]. Observ-

ing the UC Berkeley Department of Computer Science network, he made two important observations:

- The original observation of Ethernet's normal workload being lightly loaded generally still held true.
- User environments and their corresponding workstations have changed overall network utilization patterns.

It is because of this last point that we have decided to look again at the network traffic environment in 1998.

The three most visible changes that have occurred since 1990 are quite simple. First and foremost, the users have become more educated about the technology and aware of what can be accomplished with it. Secondly, the workstations have become an order of magnitude more powerful. A quick check through this author's file of computer hardware receipts shows that in 1990, a \$2000 PC consisted of an Intel 80386SX at 16MHz, 1MB of RAM, no cache, 80M of hard disk, and a 13" VGA monitor – hardly a multimedia capable system! By 1995, almost all new systems had enough power to be multimedia capable. Today, a \$500 PC outperforms it in every way by at least a factor of 10. Finally, computers have become ubiquitous commodity

items. The number workstations on the typical LAN has grown dramatically, and issuing a workstation to new employees has become as common as issuing a desk and phone.

Given these highly visible changes, a number of significant changes occurred in the basic systems infrastructure. Even the most basic operating systems have become network-aware, and what is considered a basic functionality often requires network resources. (e.g. electronic mail) Taking a closer look at common workstation we see:

- Access to a disk server (e.g. CIFS, NFS, AppleShare)
- Directory services (e.g. LDAP, NDS)
- Naming services (e.g. DNS, WINS, NIS)
- Print services (e.g. SMB, LP, Appletalk)
- Internet/Intranet access (e.g. HTTP, FTP)
- License servers (e.g. FlexLM)
- Interactive services (e.g. telnet, X)

With the growing number workstations on the network coupled with the growing demand for network bandwidth by both traditional applications as well as new multimedia applications, one would expect the average LAN to

be getting congested. In this chapter, we use Gusella's paper as a base for comparison the way he compared his work against Shoch and Hupp's paper. Specifically, we are interested in his findings regarding bandwidth consumption. We begin with a brief overview of the environment from which we took our measurements.

2.1 The CE-CERT Network Environment

The College of Engineering, Center for Environmental Research and Technology (CE-CERT) at the University of California at Riverside is a dedicated research department that serves as a crossing point for academia and commercial organizations. The group primarily focuses on alternative fuel vehicles, vehicle emissions, and transportation modeling. Their computational needs are similar to that of a traditional office – word processing, spreadsheets, web, and e-mail constitute most of the applications utilized.

The environment is very heterogeneous with 6 versions of Microsoft Windows (3.11, 95, 98, NT 3.51, NT 4 Workstation, and NT 4 Server), MS-DOS 6.22, 5 major variants of UNIX ([SPARC/x86] Linux, Solaris, SunOS, IRIX, and HPUNIX – 9 versions in total), and 2 versions of MacOS. The majority of desktop clients use either Windows 95 or Windows NT 4 for Workstations. The majority of servers are UNIX based. Several SGI and Sun workstations

are involved in heavy scientific work.

The servers are attached to multiple Ethernet switches, and the switches are connected via 100Mbit FDDI. These switches act as the top-level interconnect. The UNIX workstations are connected via another switch and uplink to the top level switches. An 80 port hub uplinks to the top level switches as well. A number of smaller 8-16 port hubs deployed at several locations in the building were connected through this 80 port hub. See figure 2.1. Much of the network topology is dictated by geography. A total of 103 hosts (including printers) are on the network of which 24 are switched. Gusella also had approximately 100 hosts.

A modified version of `ipgrab 0.6` [1] (which uses the `libpcap 0.4` library, the same as `tcpdump`) was used to set the network card to promiscuous mode and record packet headers from CE-CERT's network for 27 hours.

2.2 Summary and Comparison of Findings

Guesella's findings covered a significant amount of detail. Of this detail, we are specifically interested in comparing his findings in network utilization: how much bandwidth is being used and for what purposes.

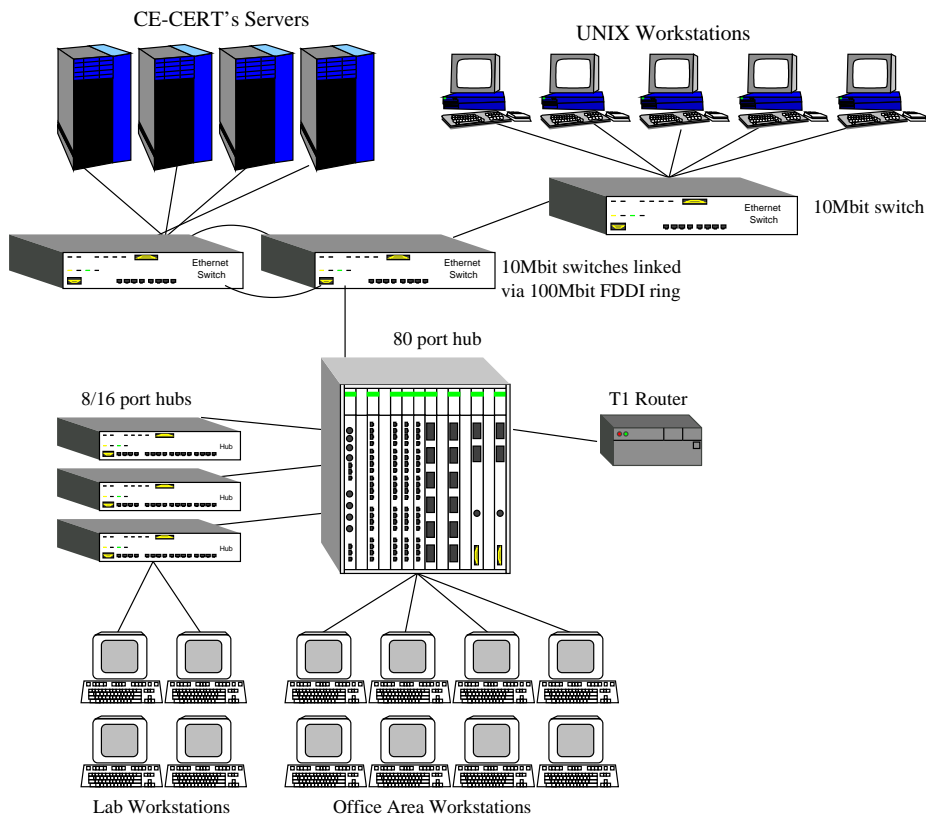


Figure 2.1: CE-CERT's Network Topology

2.2.1 General Statistics

Like Gusella, great care was taken in preparing a workstation capable of recording network traffic. The system used to collect data was a Pentium based PC with a 100Mbit 3Com 3C905 network card. The workstation had the RedHat 4.2 distribution of Linux which included kernel version 2.0.34. A modified version of `ipgrab 0.6` [1] which uses the `libpcap 0.4` library (the same as `tcpdump`) was used to set the network card to promiscuous mode and record packet headers from CE-CERT's network for 27 hours.

Although this host was connected to the network, it ran no network services such as telnet and ftp. Console logins were restricted to the root user. This host was connected to the network via the 80 port hub where a majority of the network traffic occurred.

Because the network was only 10Mbit, the host was able to keep up with all of the packets thereby achieving 0% packet loss. All packet headers were piped to the gzip compression tool and written to disk. This reduced the number of disk writes, an event which requires disabling interrupts for a short period. Since the 100Mbit network interface card was designed to accept packets at a much higher rate, we believe its additional buffer space kept it from dropping packets during those short periods when interrupts were disabled.

Two key differences exist between Gusella's network and CE-CERT's:

1. CE-CERT is primarily Windows NT while Gusella's network is primarily running variants of UNIX.
2. CE-CERT is a partially switched environment. Server to server communication as well as UNIX workstation to server communication contains traffic which we cannot observe from our observation point on the network.

Because of the segmentation in a partially switched environment, we collected fewer packets than Gusella: he collected 11,837,073 and we only collected 7,998,488. However, while Gusella's traffic was spread out over a approximately 18 hours, ours is mostly concentrated in a 10 hour time block. This reflects on Gusella's network being of a more academic nature (students still working on assignments until the early morning hours) versus CE-CERT's network more closely resembling a traditional 8 hour workday.

Overall, Gusella showed his network to average 5-10% utilization with bursts up to 20%. Our network in 1998 showed an average utilization of 10-15% with regular bursts up to 40% and higher. (See figure 2.2.)

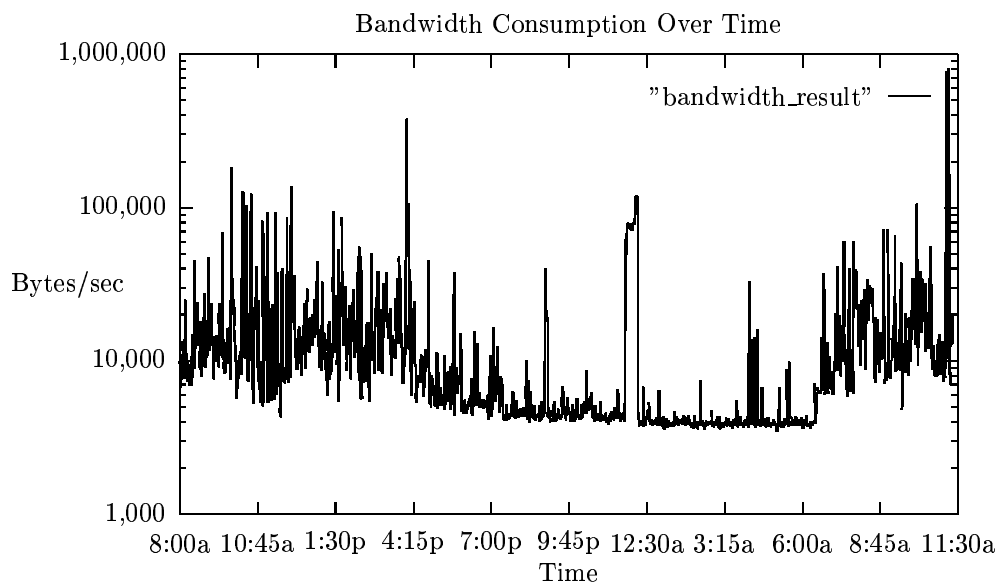


Figure 2.2: Bandwidth Consumption Over Time

2.2.2 Network Protocols

Gusella observed that his traffic could be broken down into four major categories:

1. Sun Network Disk (52%)
2. Network File System (16%)
3. General TCP (29%)
4. “Other” (3%)

This correlated with the type of hosts on his network, mostly UNIX workstations many of which were diskless Sun's. Since 1990, Microsoft Windows has made a much more significant presence on LANs pushing UNIX workstations to the minority and very low cost hard disks has eliminated the need for diskless workstations.

Given these changes, we observed our traffic broken down as follows:

1. 78% of the traffic is IP
2. 47% of the total traffic is specifically TCP
3. 31% of the total traffic is specifically UDP
4. 0.48% of the total traffic is specifically ICMP, with 99% of the ICMP packets being type 3 (destination unreachable) messages

5. 0.38% of the traffic is ARP
6. 20% of the traffic is 802.3 encapsulated. In the environment where the measurements were taken, the only platform which uses 802.3 is Microsoft Windows when it generates IPX packets. Most of the Windows to Windows IPX traffic is remote file access.
7. TCP messages sent to or from port 139, the NT SMB port (mostly remote disk access), accounted for 18.9% of the total traffic.
8. Web traffic, both sent and received, accounted for 4% of the total traffic.
9. Telnet and SSH traffic, both sent and received, accounted for 3.1% of the total traffic.
10. SMTP and POP (mail) traffic, both sent and received, accounted for 2.1% of the total traffic.
11. UDP messages to ports 659 and 1025, regularly taken by the NIS daemons on our SunOS 4.1.3 server, accounted for 2.0% of the total traffic. Compared to most ports which only accounted for 0.001% or less, this stood out. I find this to be a reasonable amount given the UNIX activity on this segment.
12. NFS barely registered in this data collection due to the primary users of it having moved to a purely switched configuration. The decision to move them into a purely switched environment stemmed from a similar

data collection experiment which showed NFS to represent 50-80% of the total packets on a regular basis. The data collected for this analysis shows the migration of NFS users onto switches has reduced network congestion considerably.

While the bandwidth utilization does not appear to be too high on this segment, we have only to look at the packet inter-arrival rates to notice something is not very right. See figure 2.3.

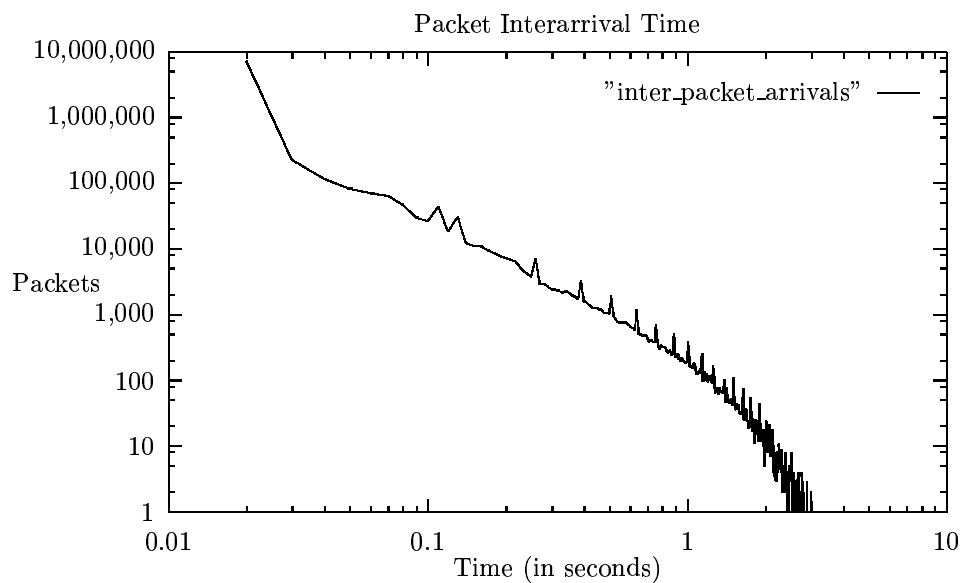


Figure 2.3: Packet Interarrival Time

Taking into account the limited resolution of the timers under this version of Linux (a problem that has since been addressed), we see almost all of the packets arrive within 10ms of one another, most of which arrive within

2-3ms. The constant stream of packets suggests a busy network. In contrast, Gusella found most of his packets arrived within 6-7ms of one another.

2.3 Analysis

In 1990, Gusella broke his traffic down into four basic types: network disk, NFS, TCP, and other. In the 8 years after his analysis, networks, workstations, and users have changed so much his divisions of traffic are almost meaningless and thus difficult to compare against. Diskless workstations no longer exist in most environments because the cost of disk per megabyte has become so low, it is possible to place a multi-gigabyte disk on what would be considered a low-end workstation. NFS, while once king of providing access to file servers now shares the limelight with SMB based protocols (Windows NT style file sharing). TCP makes up 80% of our network traffic and the only reason it is not almost 100% is because of some older Windows based hosts configured to use IPX instead of TCP for their file transfers.

A lot has changed.

2.3.1 The Migration to Windows

In chapter 1, we discussed the availability of low cost networking gear allowing businesses to construct local area networks. What made local area networks a sound business decision was the coming together of low cost workstations, the Internet's popularity, and Microsoft Windows.

In 1990, Microsoft Windows had not yet reached wide spread use it enjoyed in 1998 and thus Gusella was able to ignore the effects of its network behavior. On the other hand, we now deal with predominantly Windows based networks. This changes the type of traffic we observed versus what Gusella observed.

IP vs. IPX

IPX, originally developed by Novell in the 1980's, is one of the network protocol stacks that Windows supports. In the 1990's, Microsoft added support for TCP/IP. However, as of this writing, IPX is still the default protocol installed because it requires no configuration on the user's part. Because CE-CERT's hosts were configured and updated over several years by different systems and network administrators, many hosts still use IPX in addition to TCP/IP. Window's default behavior is to use IPX when possible.

As a result of this configuration, we saw a significant number of IPX

packets on CE-CERT's network. Although IPX can be used to contact print servers, we know it has only been used for file sharing. (CE-CERT does not have any IPX-aware print servers.)

Knowing that IPX was used for file sharing and all of the hosts at CE-CERT have support for IP networks, we can conclude that it is possible to run an almost 100% pure IP network without having to make significant changes.

Streams Replacing Interactive Traffic

The design philosophy behind Microsoft Windows has always been of one workstation for each user. As a result, the default operation for Windows is to not allow remote, interactive, logins the way UNIX allows for telnet. Additionally, since Microsoft Windows does not support X-Windows by default, fewer workstations support it. The result of this philosophy has been the sharp reduction in bandwidth consumed by interactive services. This has been replaced by more streaming style traffic caused by such things as file transfers, printing, web surfing, and audio/video broadcasts (e.g. RealAudio).

This fundamental change leads to different kinds of network traffic. In the case of highly interactive traffic, the slow nature of human typing allows for packets from different users to share bandwidth much better because packets

naturally interleave thus giving everyone a fair chance to getting their packet onto the wire. Streaming traffic on the other hand, consists of many back to back packets. Because of the capture effect, streams do not interleave well thereby making it hard to share bandwidth. The end result is users who perceive the network to be slow.

2.3.2 Faster Hosts

As mentioned at the beginning of this chapter, the average workstation in 1990 was not terribly fast. When connected to a network, one workstation alone could not consume the entire network's bandwidth by itself. By the end of 1998, we were able to take a workstation costing less than \$1500 and have it utilize a significant portion of a gigabit connection. From this information, we can derive that interpacket-arrival times will only become shorter.

Faster hosts mean more network congestion.

2.3.3 Conclusion

In the network we observed for this analysis, we saw an average of 15-20% utilization during normal business hours with bursts up to 40-45% during the day. Given the changing nature of network traffic along with the faster

hosts to generate it, we are seeing an increase in network utilization which is forcing us to think about how bandwidth gets allocated.

Many individuals are proposing bandwidth controls at either network cores through the use of ATM or at WAN gateways where bandwidth is considered most precious. But knowing that the workstations that require the greatest amount of bandwidth are at the network's edges leads us to try and exercise control at the host level as well.

Chapter 3

Quality of Service Algorithms

In an effort to improve the quality of service on networks, many algorithms have been developed to support the special handling for a variety of traffic types. In this chapter we look at two of these algorithms: CBQ and RSVP. These algorithms were chosen over others because of their general nature and because they have actual implementations available in production operating systems (most notably Linux and Solaris).

3.1 CBQ - Class Based Queueing

Class Based Queueing, or CBQ for short, is a proposal by Sally Floyd and Van Jacobson of the Lawrence Berkeley Laboratories to provide bandwidth control mechanisms at the gateway level[9]. Their proposal calls for a classi-

fication of traffic based on type of service and/or routing information. What has made the proposal noteworthy is the authors' understanding of the decentralized nature of the Internet. Their design allows for changes to occur at the local level. This gives each administrative domain control of their bandwidth and there is no dependence on both the sender and receiver to understand an out-of-band protocol to retain bandwidth control.

3.1.1 Requirements for Hierarchical Link Sharing

Links to external networks (e.g., the Internet) often must be shared between multiple organizations, protocols, and traffic types. In these situations, everyone involved in sharing the resources often wants some guarantee of bandwidth. Commercial situations often cite the need to maintain a high-speed connection for clients to a publicly accessible archive (e.g., a web or ftp site) while allowing employees to perform transactions over the link as well. In these situations, employees should not be able to saturate the link. Otherwise, client access to the web site might be blocked or become unacceptably slow, thus affecting business.

Bandwidth control to support this type of policy can be achieved using traffic shapers, such as the shaper module in Linux or the Aponet Network Manage from Aponet. This technique of bandwidth control is common and

actively in use[12]. However, limiting the bandwidth available to an application has two limitations:

1. Granular control of bandwidth allocation
2. Inefficient use of unutilized bandwidth

The granularity of bandwidth allocation is an issue in much the same way command line interfaces were an issue with non-technical users: it doesn't meld nicely with how organizations are typically structured. Because it is difficult to impose one set of rules for everyone in any reasonably sized organization, the ability to customize rules on a finer level is important.

To see the importance of inefficient bandwidth allocation, consider the situation where an organization allocates a majority of its bandwidth for client access to a web site. At night, when the web site is unutilized, backups occur through a virtual private network (VPN) with a sister office. In this case, a rigid partition of the network bandwidth would create an artificial bottleneck for the backup system, whereas the network should be available at off-peak times for transferring files over the VPN.

Given these problems, we establish the following rules for which the proposed solution must adhere to:

1. The allocation of bandwidth must be fine-tunable with a clear hierarchy.

2. Unutilized bandwidth must be shared in a formal, regular manner with streams of traffic which would benefit from access to it.

With these requirements for what should go into a hierarchical link sharing mechanism, we examine how CBQ works to accomplish them.

3.1.2 CBQ Terminology

In this section, we examine some key terms necessary to understand CBQ [9].

Classes. Each packet which enters the system must be classified based on its route and/or its type of service. (It should be noted that the meaning of the TCP type of service field is still site specific.) Each class maintains its own queue of packets and is unaware of demands on the system by other classes in the system. It is up to the scheduler (see below) to determine whether the next packet to be transmitted should come from a particular class.

General Scheduler, Link-Sharing Scheduler. Scheduling algorithms decide which enqueued packets are chosen to be delivered in which order. CBQ does not explicitly define what kind of algorithm should be used, but it does mandate that they be able to support two modes of operation: general

scheduling and link-sharing scheduling. General scheduling is a less strict mode. It allows for packets to be chosen without regard to how the gateway should be breaking down bandwidth allocation. This algorithm is used during low utilization or if only one class of traffic is in use. Link-sharing schedulers must come into play when the traffic classes are diverse and when the requested bandwidth capacity is higher than available bandwidth.

Classifier, Estimator. The process of determining which class a packet is enqueued onto is determined by the classifier. The classification process is not explicitly defined in the CBQ definition as to allow each implementation to choose their own technique [27]. suggests an efficient technique. The estimator tracks the amount of bandwidth being used by each class.

Overlimit, Underlimit, At-Limit. We use these terms to explicitly describe the state of a class in regards to its bandwidth utilization. A class that is overlimit is said to be using more bandwidth than it has been allocated. In contrast, underlimit classes are not using all of the bandwidth that have been allocated to them. At-limit classes are classes which are neither overlimit nor underlimit.

Satisfied, Unsatisfied. A class is said to be unsatisfied if it is underlimit and has a persistent backlog of packets. A class is otherwise satisfied.

Borrowing. When a class has a higher rate of packets being enqueued than packets being dequeued (though not necessarily overlimit), it can utilize bandwidth from sibling classes that are underlimit. If all of the bandwidth from sibling classes has been utilized, a class may start using bandwidth from its parent's siblings so long as they too are underlimit. This process is called **borrowing**. It may continue so long as the classes being borrowed from do not need the bandwidth themselves. Underlimit classes that require additional bandwidth can immediately take back borrowed bandwidth.

3.1.3 CBQ's Rules of Behavior

In this section we examine the rules that govern the behavior of CBQ and how they manage to provide the functionality described above.

The key to CBQ is its hierarchical design, as shown in figure 3.1. At the root of the tree, we have the entire available bandwidth. Each child node defines a class of traffic and an allotment of the total available bandwidth. Each node's children continue to sub-divide the allotment of bandwidth available to their parent.

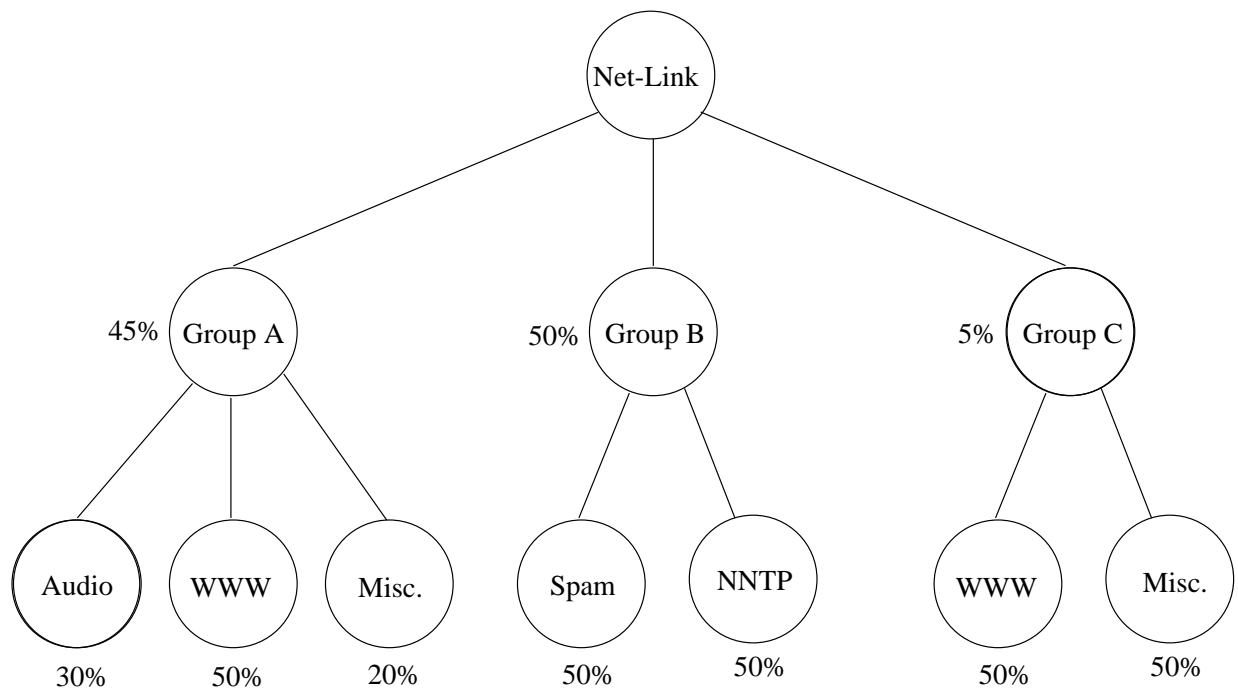


Figure 3.1: CBQ's Hierarchical Design

By providing the hierarchy, CBQ solves the first problem of being able to control the allotment of bandwidth in a well organized way. Furthermore, the hierarchical approach has an added benefit for sites that divide their bandwidth along organizational lines. Each sub-organization is allowed to make its own decisions on how bandwidth gets allocated without affecting others.

The second problem of not losing bandwidth is dealt with through borrowing.

Every class is in one of the following states at any given time:

	Underlimit	At-Limit	Over-Limit
Satisfied	Case 1	Undefined	Case 2
Unsatisfied	Undefined	Case 3	Case 4

Figure 3.2: CBQ Limits

1. The class is satisfied and underlimit
2. The class is satisfied and overlimit
3. The class is unsatisfied and at limit
4. The class is unsatisfied and overlimit

In cases 2, 3, and 4, we can possibly solve their bandwidth problem through borrowing. The process of determining whether there is another class to borrow from is as follows:

1. Check with sibling classes to see if bandwidth can be borrowed from them
2. Check with parent's sibling classes if they can borrow bandwidth
3. Continue climbing the tree checking with other sibling nodes to see if additional unused bandwidth exists somewhere, which can be borrowed.

Classes that are underlimit can become overlimit for short periods of time by bursting. This technique is a way to borrow bandwidth from others without asking first. However, it can last only for a short period of time, usually measured in terms of back-to-back packets transmitted. This can temporarily relieve congestion from a class that is suffering from a packet backlog. While it isn't a friendly tactic, it may keep a class from having to borrow bandwidth when it only needed a short burst.

3.1.4 CBQ in Linux

Alexey Kuznetsov implemented CBQ in the Linux operating system [16]. His implementation is loosely based on the NS2 [8] simulator. The code resides below the IP layer and is woven into the network layer. He can successfully divide up the traffic into classes as per specification and can perform classification by IP Type Of Service (TOS), or destination route. The transmission queue for each class is managed by multi-band priority queues.

Overall, the implementation works well. The only major limitation is its inability to accurately predict how long the system actually takes when transmitting a packet. Although this is theoretically easy to figure out based on the packet size and network speed, factors such as other system activity (e.g. significant amount of paging to disk due to virtual memory, etc.) and congestion on shared Ethernet make this difficult. Traditionally, an "End of

Interrupt” signal would be used to handle this, however, Linux does not have support for this and thus a best guess based on packet size and network speed is used. The approximation works well so long as the requested bandwidth for a class is not substantially less than the total available bandwidth for the link.

The technical details on Linux’s implementation are discussed in chapter 4.

3.2 The Resource reSerVation Protocol (RSVP)

The Resource reSerVation Protocol, or RSVP for short, is a mechanism for reserving bandwidth for end-to-end connections. By doing so, we are able to guarantee the amount of allocated bandwidth along a specific path in the network, and therefore guarantee the quality of service for the associated data stream traversing this path.

While this idea may appear somewhat obvious, the actual design of a protocol that takes into account the complexity of the Internet is not a trivial feat. Issues such as the varying amounts of available bandwidth across all hops, permission to reserve bandwidth, dynamic routing, and multicasting must be dealt with.

In this section, we discuss the capabilities of RSVP, as well as how some of the key features work. The description of RSVP in this section is based on the Version 1 Draft Specification released by the Internet Engineering Task Force [6].

3.2.1 How does RSVP fit into the system?

Considering the OSI model, RSVP works at the transport layer, and thus is capable of working on top of the IPv4 and IPv6 layers. RSVP operates in a manner similar to ICMP: it sends informational messages to other nodes, not specific processes.

A common mistake is to confuse RSVP with routing protocols. RSVP is not in charge of making decisions on the most optimal route for packets. Rather, it takes this knowledge from existing routing protocols. When RSVP has a request for a particular path, routing protocols honor the request and handle the details of providing the path.

Another common mistake is to perceive RSVP as a competitor to CBQ. It is not – if anything, RSVP is orthogonal to CBQ [7]. Their relationship is more requester/performer. RSVP requests the bandwidth, CBQ performs the necessary tasks to guarantee the bandwidth assignment once it has been

allocated.

So then, what is RSVP's place in the system? Its place is that of a higher level manager. Applications (paying customers), request bandwidth from the RSVP process. The RSVP process is responsible for determining whether:

1. The bandwidth can be allocated from the source point to the destination point, (Policy Control)
2. Determine whether the application has the necessary permissions to make the bandwidth allocation. (Admission Control).

Once the bandwidth resource is approved and allocated, RSVP turns the handling over to the packet scheduler and the application has only to send and receive data as it normally does.

To better understand the process, we examine figure 3.3.

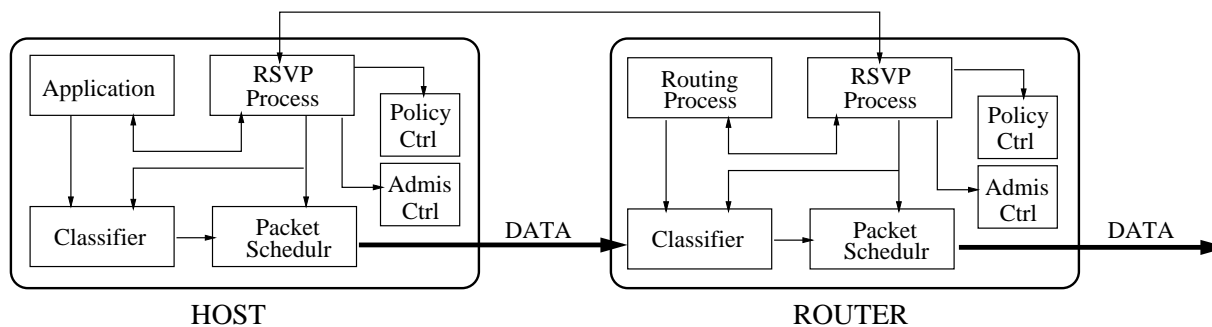


Figure 3.3: RSVP Design

In figure 3.3 we see all of the components of the system from the host to the router as described in the previous paragraph. We are already familiar with the classifier and packet scheduler functions from our description of CBQ earlier in this chapter. Notice that the data stream as depicted is unidirectional. This is because RSVP requires that each direction of a stream allocate their own bandwidth. This rule was established in anticipation of multicast situations where the sender of the stream has the privilege to establish a stream but the receiver does not.

3.2.2 How Does RSVP Work?

We begin by examining a request for bandwidth. Each request is composed of two parts: the *flowspec* and the *filterspec*. Together they form a flow descriptor. The *flowspec* simply specifies the QoS that is desired. The *filterspec*, together with a session specification, defines the set of data packets – the flow – to receive the QoS defined by the *flowspec*. The *flowspec* is used to set parameters in the node's packet scheduler, or other link layer mechanism, while the *filter spec* is used to set parameters in the packet classifier.

This request, once generated, is checked against admission control to determine whether there is even enough bandwidth available, and against policy control to check if the requestor has the necessary privileges to allocate bandwidth. Once the checks are passed, an allocation is made and the request

is sent to the next hop in the chain of routers connecting the sending and receiving hosts. When the request successfully makes it to the destination, a message is sent back to the originating process that indicates the allocation was successful. Otherwise, all allocations up until the point of failure are undone and a failure message is sent back to the requestor. Along each hop, the path is recorded so that all packets sent through this reservation are guaranteed to go through routers that know about and respect the guarantee.

When the hosts are done with the reserved connection, a tear-down message allows all intermediate routers to release their respective bandwidth allocation.

3.2.3 Key Attributes of RSVP [9]

Soft State Design. RSVP uses a soft-state design, meaning that all connections are established and torn down dynamically. Bandwidth allocation is done on a per request basis rather than hard allocating amounts for certain types of streams.

Graceful Failure. Hosts with a reservation must generate periodic RSVP “keep-alive” control messages. These messages are read by the routers along the path to the destination host informing them that the connection is still

in use. By generating these keep-alive messages, routers can automatically shutdown a connection when it doesn't see a message within a pre-determined time. This keeps hosts which have failed from holding onto a reservation.

Multicast/Broadcast/Unicast Support. The necessary mechanisms to support multicast, broadcast, and unicast connections are designed into RSVP. This is especially important for broadcast and multicast since it eliminates the need for the broadcast source from needing enough bandwidth to transmit to all of the destinations.

Support for Routers That Don't Support RSVP. The designers of RSVP were keenly aware that RSVP would not be available at all locations immediately. It is therefore possible that reservations could be made through a network that doesn't support RSVP. In such cases, a guarantee is made by those that do support RSVP, but only best effort is supported during transmissions through non-RSVP clouds. Connections are not denied in the event a non-RSVP cloud is encountered.

3.2.4 RSVP in Linux

RSVP exists as a packet classifier in the Linux QoS model. (See chapter 4 for details on the relationship between classifiers and schedulers.) As of this writing, the implementation is functional but still requiring a great deal of polish.

Chapter 4

A Tour of Linux Network Code

As mentioned in chapter 1, Linux is an open source UNIX-like operating system, which is freely available on the Internet with complete source code. Although the project has been ongoing since 1991, it remained largely a “hackers” operating system until the last 1-2 years. Today several vendors have developed installation and systems administration tools that make Linux a strong competitor to other versions of UNIX and especially to Windows NT in the small to medium server market [13].

One of the features that makes Linux so attractive is the flexibility of its networking code. Linux can be used as a gateway, router, switch and firewall all at the same time. Specific features such as IP Masquerading, which performs network address translation, and IP aliasing make it a popular alternative to commercial firewall and routing packages.

In this chapter, we examine the networking code for the 2.1.121 kernel in closer detail. The organization and specific features that are key to congestion control and QoS support will be explained. The code discussed here also reflects the structure of the 2.2.x kernel series since 2.2.x is the production release of the 2.1.x series [21].

Relation to BSD Networking

Developers new to Linux often begin with the incorrect assumption that the Linux networking code is a derivative of the BSD code. This is not true anymore. During the 1.3.x series of development kernels, the networking code was rewritten and many of the BSD-style conventions were replaced with conventions specific to Linux. This was done to achieve additional flexibility that was not easily possible with original code. The changes also provided the additional speed necessary to support very high speed interfaces, such as HPPI and Gigabit Ethernet.

A Brief Introduction to the skbuff

Those familiar with the BSD network code should be intimately familiar with the `mbuf` structure[24]. `mbuf`'s contain the contents of packets that are being processed within the system. Linux has instead adopted another structure

called the `skbuff`.

`skbuff`'s provide a more generic structure for handling packet queues. Once data has been set within the structure, it does not need to be altered when passing through the system. This is unlike the `mbuf` structure, which requires the payload to be moved when adding protocol headers. `skbuf` accomplishes this by separating header structures from payload structures.

An added benefit to this system of packet handling is the elimination of the requirement to duplicate payload content when a second copy of a packet is required. In these cases only the header needs to be copied and a pointer to the original payload is kept. Thus during normal packet processing, the payload is only copied twice: the first time from the network interface into kernel space and the second time from the kernel space to user space.

Any number of `skbuff` queues may exist so long as there is adequate memory. Functional details can be found in the Linux source tree, in files `linux/net/core/skbuff.c` and `linux/include/linux/skbuff.h`.

4.1 Network Code Organization

Linux follows the OSI layering model for its network code. This allows Linux to support protocols other than the defacto standard TCP/IP. These protocols include Appletalk, IPX, AX25, and even the next generation of TCP/IP, IPv6. The OSI model specifies 7 layers, as shown in figure 4.1. However, the top 3 layers (session, application and presentation) are actually handled by user processes rather than the kernel. The bottom layer (physical) is also out of the kernel's scope. Therefore, the kernel only needs to deal with three of the 7 layers: the data link, network, transport, and session layers as seen in figure 4.2. The lowest layer is actually a pseudo-layer where the device drivers connect the physical layer to the data link layer [25].

To understand the path that a packet may take through these layers we examine figure 4.3, which shows how TCP/IP fits into this model. In the following sections, we examine each of the layers, their relation to TCP/IP, and some specific details about Linux's implementation. Our goal here is to not only shed light on the internals of Linux's TCP/IP stack, but also to establish a foundation for understanding why and where we made modifications for this thesis.

Before we dive into the functional details, we present figure 4.4 which shows which functions are in which layers and the order in which they com-

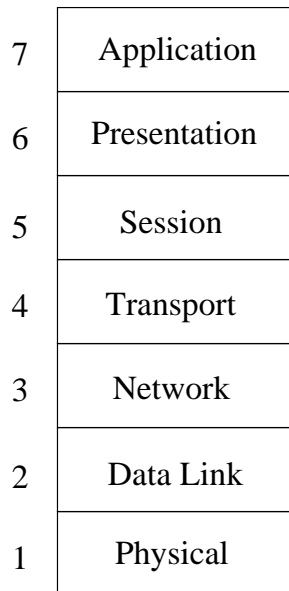


Figure 4.1: OSI 7-layer model

municate with one another. This is simply a reference diagram for the remainder subsections.

4.1.1 Device Drivers

Device Drivers are the interface between the hardware and the kernel. This software controls a variety of network interface cards (NIC) currently available on the market. Since each NIC provides a different interface and command set, separate drivers are required for each vendor's product family. However, a skeleton driver written by Donald Becker continues to serve as a basis for most drivers. (see the file `linux/drivers/net/skeleton.c` in the

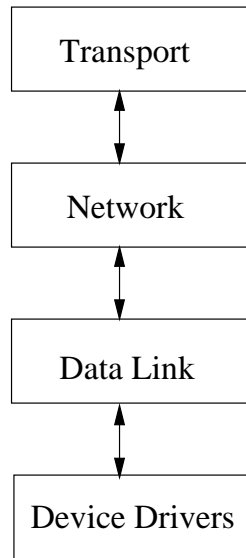


Figure 4.2: Linux 4-layer model

Linux source tree.)

For every network card that is recognized and supported in the system, a `struct device` entry is created. Calls to the driver are done by referencing the structure, thus providing the kernel with a standard interface to configure the device as well as to send and receive packets. More specifically, each network interface driver must provide functions to:

- open the device (`dev->open`)
- close the device (`dev->close`)
- send a packet (`dev->hard_start_xmit`)
- provide statistics (`dev->get_stats`)

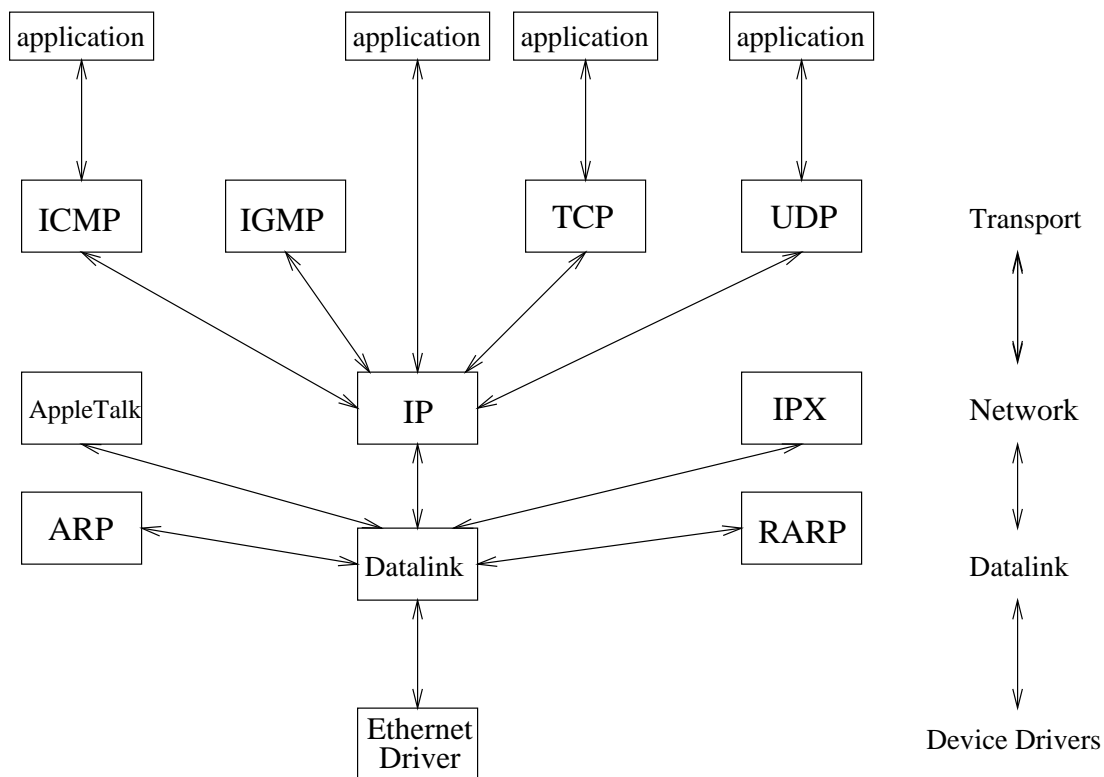


Figure 4.3: How TCP/IP maps into the OSI model

- set the multicast filter (`dev->set_multicast_list`)

When a NIC accepts a packet on the wire, it has only one choice in going up the protocol stack – calling the `netif_rx` function (`linux/net/core/dev.c`) in the network layer. In contrast, the network layer may accept packets from many different network interfaces.

Someone curious about NIC drivers is recommended to start with the loopback interface driver which is at `linux/drivers/net/loopback.c` or the skeleton device driver.

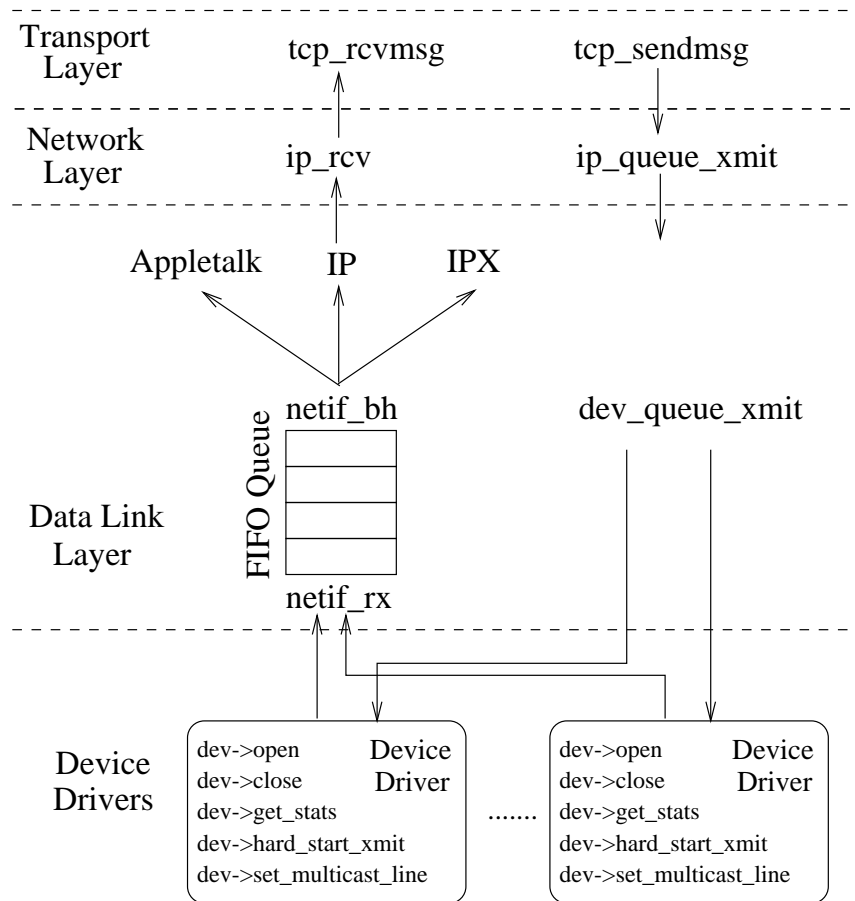


Figure 4.4: Relationship between actual Linux functions and the OSI model.

4.1.2 The Data Link Layer

In the OSI model, the purpose of the data link layer was to determine which protocol was to be used for incoming packets and to enqueue outgoing packets to the appropriate network interface. Linux performs these tasks and others in the `linux/net/core/dev.c` file.

Receiving Packets from the Device Driver

Incoming packets are handled through a queue that mediates a classic producer/consumer relationship between the functions `netif_rx` and `net_bh`. The reason this was done was so that the kernel would block for the shortest amount of time necessary during the enqueue process. (Since it is possible for multiple network interfaces to try to enqueue a packet at the same time, it is necessary to block.) This structure maximizes the amount of concurrency that can be achieved on the input side.

When a packet is received, the `netif_rx` function is called. The purpose of this function is to determine if there is enough space to enqueue the packet on the incoming FIFO queue. If there is not enough space, the packet is dropped. Appropriate counters are updated for statistics tracking.

A separate routine (`net_bh(void)`) processes the queue. Once inside of `net_bh`, the packet is dequeued and handled as follows:

1. A check for “fast routing” to support packet forwarding in situations where there is only one routing path and thus the packet does not need to be handled by upper layers but rather immediately resent.

2. A check for network bridging is done for the same reasons as fast routing.
3. Finally, the packet protocol ID is matched against known protocols and the packet is passed up to the appropriate handler. This is a one to many relationship.

Sending Packets to the Device Driver

Outgoing packets are passed to `dev_queue_xmit`. This function is responsible for determining which QoS mechanism is responsible for scheduling the packet's actual transmission. The packet is then passed to the appropriate QoS mechanism. How the QoS mechanisms work are discussed in further detail later this chapter.

Once an outgoing packet is enqueued, a function to “run the queues” is called. Running the queues is the process of iterating through all of the current outbound queues to determine whether a packet from a particular queue needs to be transmitted. In order to support different QoS algorithms, this function is abstracted so that each QoS mechanism can use its own queueing function. The default “QoS” mechanism is a FIFO queue.

4.1.3 The Network Layer

The network layer is where the protocols such as TCP/IP, IPX, and Appletalk begin their work. It is entirely possible for multiple protocols to exist within a single system and to work independently of one another. One of Linux's greatest strengths is its ability to work in environments where many protocols are in use and to serve as a means of bridging between them.

IP itself is actually the foundation for higher layers where TCP, UDP and ICMP reside. Although it is possible for user processes to work with IP directly, it is uncommon. This is because IP alone only provides connectionless and unreliable delivery of packets [26].

We define a *connectionless* protocol to be one that treats each datagram independently. This requires every datagram to be buffered, sent, received and routed regardless of the datagrams in front of it and the datagrams behind it. Because of this behavior, it is entirely possible for two consecutive datagrams to take different paths between the same end points. If the paths have different lengths and/or latencies, then the datagrams may arrive out of order. This is valid behavior for IP – its task is only to try to get the packets to the destination. Ordering issues are outside its scope, to be handled by higher layers such as TCP if needed.

We define *unreliable* protocols as those that do not guarantee the delivery

of their datagrams. With IP, a best effort is made to deliver the datagram. However it is quite possible that a datagram will not reach its destination for some reason (e.g. buffer overflow on an intermediate router). Such transient errors are silently ignored by the IP layer. Permanent errors, such as an invalid destination address, may trigger the router to return an ICMP “Unreachable” message back to the sender alerting it of a failure [24]. Detection of a lost datagram and initiation of a retransmit are the duty of higher layers.

The implementation of the IP layer in Linux complies with RFC standards [2]. This includes the ability to:

- Perform sanity checks on the packet. (e.g. CRC validation)
- Perform routing based on destination IP address.
- Based on the specified protocol in the IP header (see figure 4.5, “8-bit protocol”), pass up the payload to an appropriate higher level protocol or user process.

In addition to these basic functions, Linux also comes with the ability to:

- **Perform IP Masquerading** A feature which allows a single host to act as both a router and network address translator (NAT) such that it is possible for a private LAN to access an outside network with a single IP address. Linux tracks individual connections between the

masqueraded hosts and destination hosts, and performs transparent routing between them.

- **Perform IP Firewalling** The TCP/IP stack can be configured to deny (do not return ICMP unreachable message), reject (return ICMP unreachable message), or accept incoming, outgoing, or masqueraded packets based on IP address, port number, or TCP header flags. Read the IP firewall source code [22] for further details on this feature.
- **IP Rules** IP Rules are usually routing rules which allow Linux to act on behalf of a host that is located in a masqueraded subnet. Thus, a site that has two IP addresses allocated to it through one ISDN connection can make the Linux host answer both IP addresses. Linux, when seeing packets destined to the second address can automatically forward packets to a private network. In addition to routing rules, IP Rules can perform IP header manipulation based on certain criteria. For example, packets destined to a particular host may have its Type of Service (TOS) field set to a special value for Quality of Service purposes.

The basic functions of IP are well explained in Stevens [24]. Although Stevens discusses the 4.4BSD-Lite release, the concepts remain the same. Further reading is also available in RFC 1122.

Our interest in these additions to the IP layer stems from their expanding

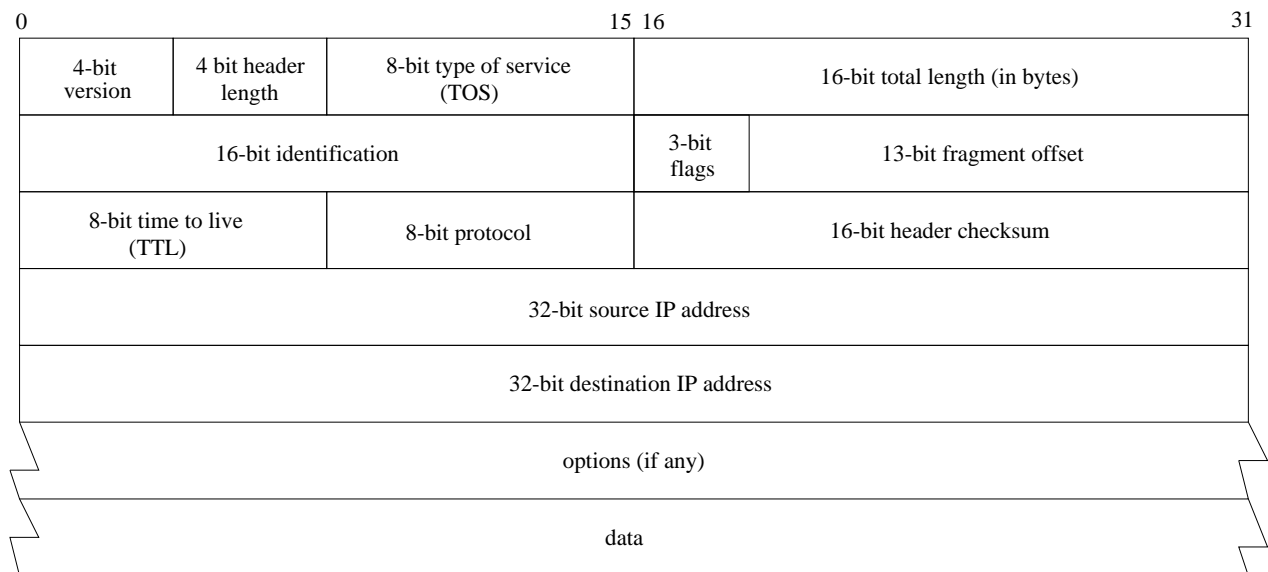


Figure 4.5: IP Header

use amongst the Internet community. The ability to attach an ADSL, ISDN, or even a PPP connection to one host and then have it shared amongst an entire office without needing to purchase a dedicated router is appealing to Network Administrators.

But as with any shared resource, the problem of making all the users share it fairly becomes an issue. Posts to the Linux-Net mailing list regularly showcase sites that have configured their Linux systems to share a low bandwidth connection with an office only to find one or two users consuming the link with large file transfers. We see here that the ability to control certain types of streams from consuming a link would be a very useful feature.

In between Layers: ICMP and IGMP

ICMP and IGMP are special protocols that exist in a layer above IP. However, their properties do not make them appropriate to classify in the transport layer. For discussion purposes, we group them in the transport layer but consider them adjacent to IP.

ICMP (Internet Control Message Protocol) is a host to host message protocol used to gain status information about the network. The most common message type is “Echo”, which is used by `ping` and `traceroute`. A full list of ICMP messages can be seen in Stevens [26] on page 71.

IGMP (Internet Group Management Protocol) is a host to host message protocol used to establish multicast groups. These types of messages are rare. However support must exist in the TCP/IP stack for it to be RFC compliant.

How IP Receives Packets from the Network Layer

The main IP receiver routine is `ip_rcv` in the `linux/net/ipv4/ip_input.c` file. The function does the following with each packet received:

1. Determine if the NIC is in promiscuous mode or the packet is destined for this host. If not, stop analysis and drop the packet.

2. Check that the packet complies with RFC 1122. Drop the packet on failure. (e.g. bad checksum)
3. Check routing tables, firewalls, masquerading entries and rules. Handle packet accordingly.

If a packet is destined for the receiving host (i.e., we are not masquerading), it is passed up to the `ip_local_deliver` function which takes care of delivering packets to the higher level protocols or to a raw socket.

How IP Sends Packets to the Data Link Layer

When a higher level protocol is ready to send a packet through IP, it calls the `ip_queue_xmit` function in the `linux/net/ipv4/ip_output.c` file. The function does as follows:

1. Verify that a route exists for this packet.
2. Construct IP header
3. Determine if the packet needs to be fragmented. If so, go to another handler.
4. Compute checksum and transmit.

Unroutable packets are dropped at this point.

4.1.4 The Transport Layer

As the name implies, this layer understands the concept of an end-to-end data transport. We define a session to be communications between two processes from source to destination host rather than just two hosts. In the TCP/IP protocol stack, this is done through the use of *ports*.

In TCP and UDP, an additional layer of headers is added to the already existing IP header. These headers (see figures 4.6 and 4.7) allow a process to specify a source and destination port. The kernel is then responsible for providing a mapping between processes and port numbers. Because of these port numbers, both TCP and UDP offer an additional level of multiplexing compared to IP. When a packet is received at the destination, the receiving process knows not only from which host the packet originated, but also which process on that host created it. This allows multiple processes on both the sender side and receiver side to communicate with one another independently.

TCP offers the additional functionality of delivering datagrams in a reliable, ordered manner. This is achieved by doing the following:

- TCP breaks each message down into what it considers to be optimum sized segments.
- When a datagram is sent, TCP maintains a timer to find how long the destination took to respond. If the response takes too long, TCP

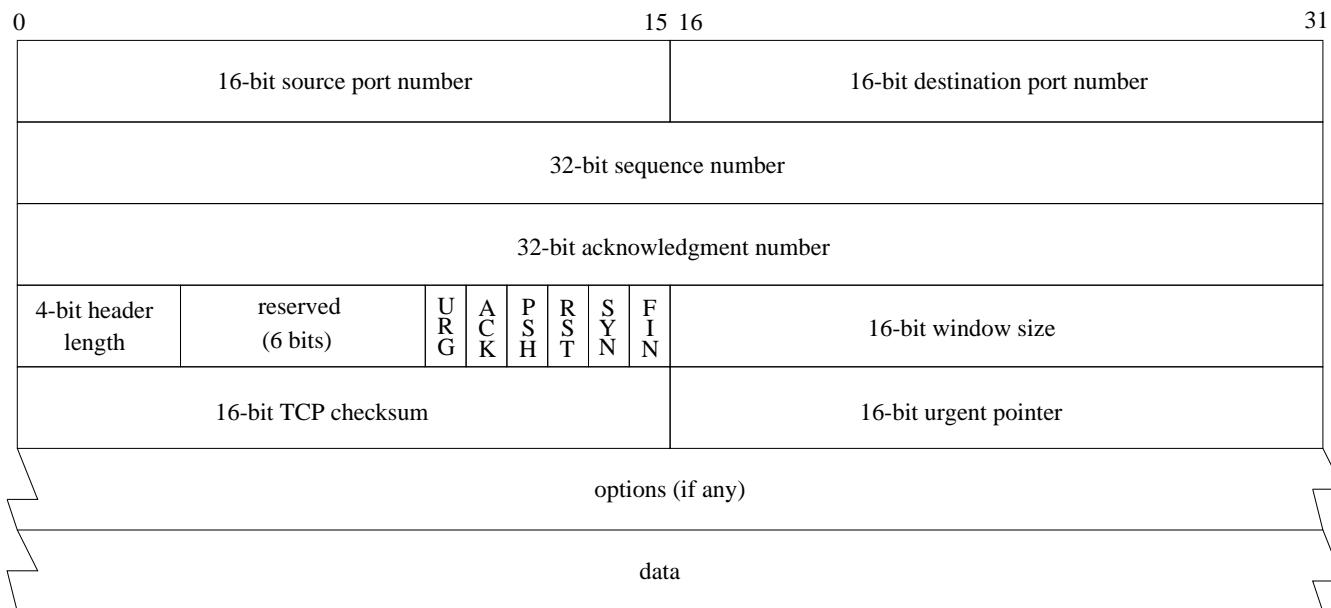


Figure 4.6: TCP Header

assumes there was an error and resends the datagram.

- When TCP receives a datagram, it sends an acknowledgement. (TCP attempts to optimize this process by batching as many acknowledgements as it can and piggybacking them on reverse traffic.
- TCP maintains a checksum of each datagram. The receiver also computes a checksum and compares the two values. Mismatches indicate an error and bad packets are disposed of. There is also a separate checksum on the header, so you can complain about corrupt data. (This does not work if an addresses error misdirected someone else's packet to you.)

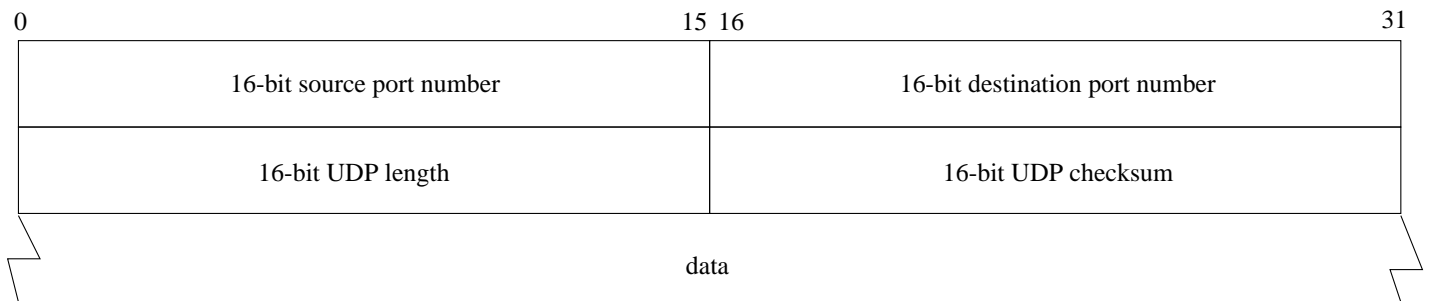


Figure 4.7: UDP Header

- Since TCP uses IP to send and receive datagrams, it knows that IP datagrams may arrive out of order. TCP puts out of order packets aside until all of its predecessors arrive. This allows TCP's client process to safely assume it will receive its stream in the order it was sent.
- TCP automatically throws away duplicate packets.

Linux does not change the behavior of TCP. Small changes in the code are typically for the purposes of tuning and SMP support. The key functions that handle TCP's behavior can be found in `linux/net/ipv4/tcp_ipv4.c`, `linux/net/ipv4/tcp_input.c`, and `linux/net/ipv4/tcp_output.c`. The primary send routine is `tcp_v4_sendmsg` and the primary receive routine is `tcp_recvmsg`. All of the high level entry routines are specified in the `struct proto tcp_prot` structure in `linux/net/ipv4/tcp_ipv4.c`.

4.2 Congestion Control

As the number of users on a network increases, issues of network congestion must be dealt with. TCP/IP has four built-in mechanisms to try to control congestion on a connection by connection basis. In this section, we examine them in closer detail.

4.2.1 Delayed Acks

Delayed acknowledgements are a mechanism for reducing unnecessary packets. However, as a side effect, reducing the number of packets also helps in controlling congestion.

Because TCP is tasked with making sure data arrives to its destination, the receiver needs to return positive acknowledgements (ACK's) to the sender, which indicate that each packet has been received. Otherwise the sender could not be sure that the datagram was received correctly and would be unable to delete its copy. However, if TCP were to generate a separate ACK for every packet it receives, then the network would quickly become overly congested. This is especially true in half-duplex networks where the immediate transmission of an ACK would likely collide with the next packet in a stream thus forcing both hosts to retransmit.

To use acknowledgements in a more efficient manner, each packet received is not immediately acknowledged. Instead, each packet successfully received is noted. The system waits for a packet destined to the host that should receive the acknowledgements and when such a packet is found, the acknowledgement information is marked inside its TCP header (see figure 4.6). By doing this, we save ourselves from having to send two separate packets: the reverse packet destined to the original sender and a second packet for carrying the acknowledgement. This technique is called piggy-backing because two purposes are served by one packet.

An additional bonus for using delayed acknowledgments is being able to acknowledge multiple packets with a single cumulative ACK. This works because a packet with the ACK bit set signifies that all of the data up to the specified point in the sequence have been received. For example, if two back-to-back packets arrive from a single host, then it is only necessary to acknowledge the data up to the second packet. By doing so, TCP implicitly acknowledges the data in the first packet. Cumulative ACKs also provide added robustness, since the data in a lost ACK will be automatically included in the next ACK.

Obviously, there would be a problem if an ACK were to wait indefinitely for another packet going the same way. This could lead to stalling or even worse, deadlock. To make sure this doesn't happen, the TCP specification

requires that an ACK not wait longer than 500ms. After 500ms, the TCP stack must generate an additional packet to transmit the ACK.

The Host Requirements RFC [2] states that TCP implementations *should* include delayed acknowledgements. Linux does include support for delayed acknowledgements. The RFC also requires that if delayed acknowledgements are implemented, then the timeout for generating a separate ACK must be less than 500ms. Linux complies with this requirement by using a timeout delay of 200ms. (See `linux/net/ipv4/tcp.c` in the kernel source tree.)

4.2.2 Nagle's Algorithm

In the same vein as Delayed Acknowledgments, Nagle's Algorithm works to reduce congestion indirectly, by sending packets more intelligently.

In an interactive TCP connection, most packets contain only one byte of payload data. The packet size so far is 41 bytes: 20 bytes for the IP header and another 20 bytes for the TCP header. A few additional bytes for the data link layer header need to be added as well. (RFC 894 Ethernet needs 18 more bytes [26] for a total of 57 bytes.) These extremely small packets are called *tinygrams*.

Tinygrams are acceptable in low-latency/uncongested local area networks where they are unlikely to cause congestion. (Interactive sessions, by definition, are run by humans who are many orders of magnitude slower at injecting information into a network than, say, a process handling file server requests.) On the other hand, high-latency and wide area networks are far more prone to being congested with such packets. Enter Nagle's Algorithm.

Nagle's Algorithm [19] states that when there is data that has been transmitted but not yet acknowledged from the receiver, the sender must not transmit any small segments (tinygrams). Instead, the sender should enqueue the small payloads. When an acknowledgement is received, all of the small payloads are put into a single packet and transmitted.

What makes this algorithm particularly interesting (and effective) is its self-clocking nature. When acknowledgements arrive quickly, transmission occurs quickly. Ideally, this makes its presence in a LAN environment unnoticeable since it rarely if ever has to enqueue payloads for delayed transmission. (Stevens calculates that a human would have to be able to type at over 60 characters/second over an uncongested 10Mbit Ethernet to be able to trigger Nagle's Algorithm [26].) In WAN environments however, it automatically begins working since the ACK packets are more likely to arrive at a slower rate due to congestion.

The Host Requirements RFC [2] recommends all TCP implementations to have Nagle's Algorithm.

Disabling Nagle's Algorithm

Nagle's Algorithm bases its timing mechanism on the rate at which it receives ACK packets. Delayed Acknowledgements work by artificially delaying acknowledgements in order to minimize network traffic. It doesn't take a great deal of imagination to find instances where these two mechanisms will work against each other.

Consider the interactive remote login session. Here, several characters entered by the user for transmission may be delayed for transmission while waiting for the server to send an acknowledgement. At the same time the server may be waiting for additional characters to echo back so that it may piggy-back the acknowledgement. The result: sporadic delays experienced by the user [26].

In order to compensate for these kinds of situations, the host requirements RFC [2] requires that if an implementation of TCP has Nagle's Algorithm, it must be possible for a user process to disable it using the `TCP_NODELAY` socket option.

4.2.3 TCP Sliding Window

A system must have a buffer for storing incoming packets in a queue for processing, especially if it is taking the system longer to process each incoming packet than the minimum interval time. It is of course desired that this buffer not overflow. After all, if the system is receiving more packets than it can handle, it is safe to assume that the network is congested. Forcing someone to resend a packet only makes the congestion worse in this case.

In order to limit the impact of congestion, TCP uses a special kind of buffer called a Sliding Window[2]. Looking at figure 4.8 we see a stream of packets numbered 1 through 7. (In actual fact, TCP sequence numbers count bytes, rather than packets. However, packets are always expanded to the Maximum Segment Size, or MSS, to avoid the Silly Window Syndrome explained below. Hence, to simplify this example, assume that these sequence numbers represent multiples of the MSS.) As each packet is read from the network, the right side of the window expands. As each packet is processed by the system (sending acknowledgements, passing the data back to the user process, etc.) the left side of the window moves to the right as well.

This window has a maximum size indicating how much buffer space is allocated to queueing incoming packets. When the buffer is full, additional packets cannot be read and therefore must be dropped. To avoid having to drop packets, TCP uses *Window Size Advertisements* as a way of informing

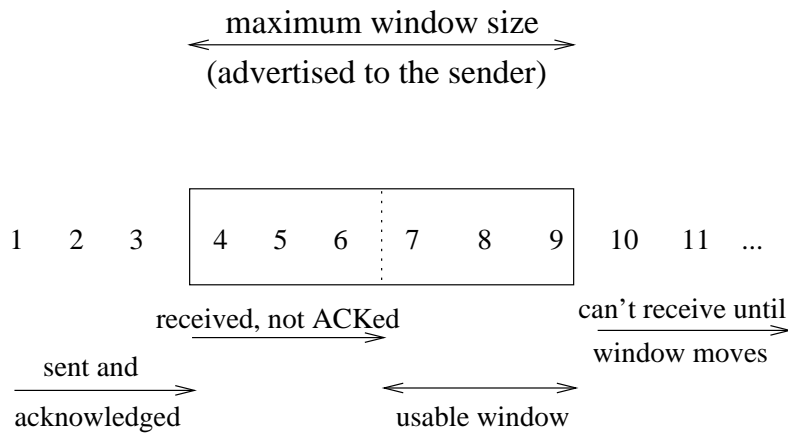


Figure 4.8: TCP Sliding Window at the Receiver

hosts communicating with it as to how much buffer space is left.

By controlling the window size, a receiver can control the rate at which other hosts send data to it. When a host is congested, it can advertise a window size of 0 to force other hosts to stop sending until they receive future advertisements of available buffer space. Ideally, window size advertisements piggyback with other packets to avoid generating additional network traffic. This tends to work well in real life since extra packets are only required after the receiver has announced a window size of 0.

Silly Window Syndrome

Silly Window Syndrome, or SWS, is the condition in which small amounts of data are exchanged across the connection instead of packets containing the

maximum segment size. [3] This can be caused by either end of the connection: the sender can transmit a small packet instead of waiting for additional data, or the receiver can advertise a window smaller than the MSS. The result of this behavior is drastic changes in the window size in the middle of the stream. The cause of these drastic changes in window size is because the receiver will see very little new data and be able to process everything in its queues. The receiver will be able to open its window size enough to allow the sender to send a large number of back to back packets which fill up the receivers' queue very quickly. This situation results in very poor performance.

In general, TCP should do one of two things to avoid SWS: the first is not to advertise a window until the window size can accommodate one complete segment as defined by the MSS or one half of the total receiver space is freed, whichever is smaller. The second thing is on the sender's side. The sender should avoid transmitting until one of three conditions are met:

1. a full MSS can be sent
2. we can send $\frac{1}{2}$ of the largest segment size ever advertised
3. we send everything we have while we are not expecting an ACK or Nagle's algorithm has been disabled.

The host requirements RFC [2] requires that a SWS avoidance algorithm be in place for both sending and receiving. Linux complies with this requirement.

4.3 Quality of Service in Linux

Quality of Service mechanisms arrived in the Linux source tree during the 2.1.x kernel series. These mechanisms included: Class Based Queueing (CBQ), Clark-Shenker-Zhang (CSZ), Random Early Detection (RED), Stochastic Fairness Queueing (SFQ), Token Bucket Filter (TBF), and Resource reSerVation Protocol (RSVP). Our focus for this thesis is on the implementation details of the CBQ algorithm. However, many of the interface guidelines apply to the other algorithms as well.

In this section we will discuss where in the network stack QoS was implemented, the separation of the classifiers, schedulers, and queues, and how they achieve reasonable performance through standard API's.

4.3.1 Where does QoS Fit?

Although we are focusing on the TCP/IP stack implementation in Linux, it should be noted that Linux also supports other network protocols. The two

most frequently used other protocols are IPX and Appletalk, which allow Linux to communicate with other Novell and MacOS based systems, respectively. Hopefully it will not be too long before IPv6 can be added to the “frequently used” list.

When Alexey Kuznetsov, one of the key developers of the Linux networking code, decided to implement QoS algorithms, he made sure that his implementations would not fail in the event that non-TCP/IP protocols are being used. To accomplish this, QoS is implemented in the data link layer. As we can see in figure 4.9, this allows traffic generated by all of the protocols to pass through the QoS code and thus be accounted for.

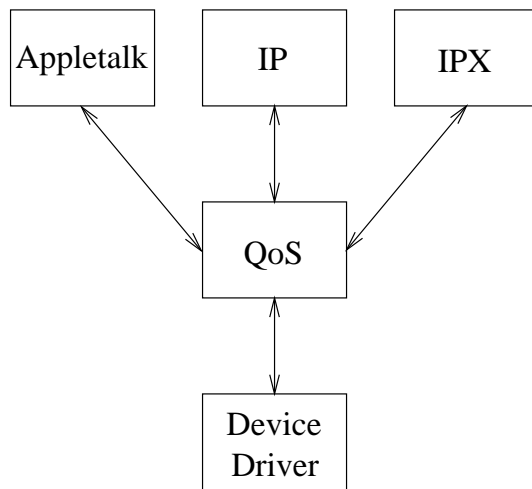


Figure 4.9: Where QoS fits in the network stack

4.3.2 Separation of the Classifiers, Schedulers, and Queues

Up until this point, we have only mentioned scheduling algorithms such as CBQ. However, scheduling algorithms do not work alone – they require the assistance of a classifier and a queue manager.

The differences among these three systems are subtle, but of course important. Their relationship is best summarized in figure 4.10.

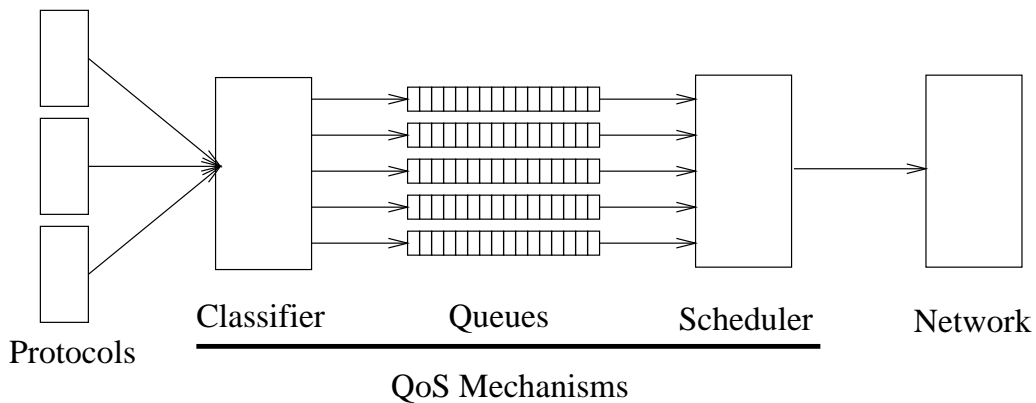


Figure 4.10: The relationship between classifiers, schedulers, and queues

The classifier has the job of determining into which queue each packet should be placed, based on user-specified rules. Different classifiers can be optimized for different purposes. Linux currently has four classifiers to choose from: a routing-table based classifier, an RSVP classifier, an RSVP for IPv6 classifier, and the “Universal” (generic) classifier.

Queue managers provide the necessary services to manage the various queues used by schedulers. As you may recall from the discussion of CBQ in section 3.1, there is no specified algorithm for how queues should be managed, and therefore this technique of separating schedulers from queue managers is a natural extension. Simple FIFO and priority queues are the only choices offered as of this writing.

Finishing with the far right of figure 4.10, we see the scheduler, which is the core of the Linux QoS system. The scheduler has the task of deciding which packet in its many queues should be transmitted next. Packet selection depends on which algorithm is in use. The various algorithms available were discussed at the beginning section 4.3.

How Separation Affects Performance

Concerns over performance issues may lead some to question the notion of keeping the three systems abstracted from one another. As we have learned from early implementations of TCP/IP, care must be taken to minimize memory to memory copies and other such CPU-consuming tasks.

The developers solved this issue by taking a C++ class approach. Each classifier, scheduler, and queue manager has well defined functions that it must offer much like public functions of a C++ class. Additional support

functions may exist. However, they are not to be directly called by other modules. By having done this, each module is interoperable with other modules.

Memory to memory copies are eliminated through the use of `skbuff`'s. Function calls to different modules only required pointers to the appropriate `skbuff`. If a `skbuff` needs to move from one queue to another, only some link list manipulations are needed.

Although no performance tests have been run to compare integrated vs. non-integrated approaches, it is generally agreed by the development community that the current technique is efficient enough and the benefit from maintaining the abstraction is simply too good to ignore.

4.3.3 Linux Specific Details

The glue that holds the QoS system together is the scheduler. The scheduler serves as the interface to the queue manager and classifier functions. This is established during the scheduler's initialization procedure.

When a scheduler starts up, it must first register itself with the system using the `register_qdisc` function call. The system requires that the scheduler support a well defined set of routines to perform queue manage-

ment (e.g. enqueue, dequeue), class management (e.g. get class, put class), and scheduler management (e.g. initialize, reset). This is specified through two structures, `struct Qdisc_ops` and `struct Qdisc_class_ops`.

The `struct Qdisc_ops` contains a pointer to the `struct Qdisc_class_ops` which has pointers to functions to the class manager. Each class manager is specific to a scheduler. The rest of the `struct Qdisc_ops` is pointers to functions that take care of the queue management and general scheduler management.

All user-configurable parameters, such as establishing classes, selecting queueing algorithms, and tuning scheduler parameters is set through a userspace tool called `tc`. Using `tc`, schedulers can be associated with network interfaces and classifier rules can be setup.

Sending a Packet

When a packet is passed into the network layer for transmission from the transport layer, the `dev_queue_xmit` function which is in the `linux/net/core/dev.c` file. This function then calls the queue manager for the device. The default queue manager for each device is a single FIFO. If another queueing discipline such as CBQ is associated with the destination network interface, CBQ's enqueue function will be called instead.

Assuming CBQ is the queueing discipline for a packet to be transmitted, its enqueue function will first call the packet classifier to determine which class the packet should fall under and then enqueue it. Once the packet has been placed in a particular queue, a function is called to activate that class (`cbq_activate_class`). This sets a flag indicating that the class has something to transmit.

Whenever a packet is sent, `qdisc_wakeup` is called from `dev_queue_xmit` to give the scheduler a chance to send a packet (not necessarily the packet being enqueued). It is not necessary that the scheduler send a packet though. In the case of CBQ, for example, all of the classes may be overlimit and should not transmit anything.

All of the code for schedulers, classifiers, and queue managers can be found in the `linux/net/sched` directory of the Linux source tree.

Chapter 5

Implementing Rate Control

For this thesis, several extensions to the network code in the Linux kernel were implemented. In this chapter, we explain each extension, its purpose, and how it was done.

5.1 Keeping Track of Rates

In order to provide rate control on a per-connection basis, code needed to be added to the TCP/IP stack to keep track of the rate for each open socket connection in packets per second. To accomplish this, we had to keep in mind several key issues:

1. The rate tracking mechanism has to introduce extremely low overhead.

2. Because history data for each socket needed to be maintained within the socket structure, it needed to remain small.
3. The process of actually calculating the rate based on data collected needed to be accomplished using integers only (kernel limitation).
4. Any processes to track this information would have to be done at the TCP layer since it is the first layer that tracks individual connections.

5.1.1 “Chunkifying”

In order to solve the problem of creating a low overhead mechanism for tracking each packet, we decided to use time stamps since the kernel has to keep the counter up to date anyway. As each packet arrives into the TCP layer, `tcp_data_queue` is called to queue delivery to the user. As soon as the packet is appended to the receive queue, the current time is noted and a counter is incremented.

We begin with a user-specified time frame, in which we accept all incoming packets and count the number of packets received. Because it is possible for the actual amount of time elapsed to be longer than the user requested time frame, we scale the packet count down by multiplying the user specified time frame with the actual packets received and then dividing by the actual time it took to receive the packets. For example, if the user specified time is set to 50000 microseconds and we track 10 packets in 55000 microseconds,

we scale the 10 packets down to $((50000 * 10) / 55000)=9$. This tells us we are receiving data at a rate of 9 packets per 50000 microseconds.

We call this block of packets received in the user specified time frame of 50000 microseconds a “chunk”. We keep track of a fixed number of chunks. However, the time frame each chunk represents is user configurable at run-time through `/proc`.

To keep the amount of history information constant, chunks were kept in a circular buffer. The number of chunks stored in the circular buffer is established at the kernel’s compile time. Because a chunk could represent time slots as large as 50ms, several seconds worth of rate information could be easily kept.

“Chunkifying”, as we have come to call this technique, satisfies the first two requirements we noted earlier: It has very low overhead since the time computations are minimal (Compiling to assembly with `gcc -S` under the Pentium architecture shows the math computations to take less than 30 instructions), and the small array of integers to store the historical data requires little memory.

5.1.2 Computing the Rates

Typically, an incoming rate is computed by setting a timer and counting how many packets are received before the timer goes off. In our case, the possible number of simultaneous connections on a busy system made setting timers for every connection a bad idea since the system could easily become swamped with dealing with so many timer interrupts. Instead, we opted to compute the rate whenever a packet arrived. Since our goal is to control a congested system and leave an uncongested system alone, this philosophy worked nicely.

When a packet was received, it was accounted for, “chunkified”, and then tested to see whether or not a chunk had been completed or not. (A chunk was completed when the first arrival occurred after the end of the user-specified time interval.) When a chunk was completed, we scanned the circular buffer of “chunk data” and computed the average number of packets per second. This value was then set in `sk->computed_rate`, where `sk` is the socket structure for each particular connection.

Observing the code’s behavior, a straight average appeared to work well enough and required the least amount of computation which satisfies requirement 1. Simple trials showed the mechanism did a reasonably accurate job of keeping track of rates.

5.1.3 Aging Rates

The following scenario presented a problem to the above solution: a socket experiences a burst of traffic followed by an abrupt silence. In this situation, a high rate would be computed and then stored in the socket’s “current rate” variable. But when no new packets would arrive on that socket, the rate would not be updated to reflect the following silence. This situation would leave the system with a false impression of the data throughput.

To solve this problem, a kernel timer was set to go off every two seconds. This time would invoke a function to scan the open sockets and check for situations where there has been no new traffic for a user specified time interval. If there has been no traffic, the history buffer is zeroed out and the “current rate” variable is set to zero as well.

5.2 Controlling the Rate

In chapter 4 we discussed the advertised window size, which is sent from a receiver to a sender to indicate how much data should be sent at one time.

We agreed in chapter 1 that it may be possible to control a sender’s transmission rate by manipulating the advertised window size. In order to determine the validity of the claim, we used the rate calculation engine de-

scribed in the previous section as a guide for whether to shrink or expand a window for the sender.

5.2.1 Handling TCP

By modifying the function that computes the advertised window size (`__tcp_select_window`), we were able to adjust the window size based on the computed rate for the given socket rather than just on the available buffer size.

The general algorithm for determining whether or not the window size needs to be reduced and if so by how much is very simple. When a socket begins going over the maximum user specified rate, the window size begins shrinking. Windows over 8k in size are reduced substantially because the change in rate per 1k of window size over 8k tends not to be too drastic [24]. As the rate falls below the 8k marker, the window size drops at a lower rate until it reaches the 2k mark at which time it drops at a 256 byte rate.

When a window size is being computed to send to the sending host, the current socket rate is compared against the user specified rate. If it is necessary to try to change the rate, the window is scaled reduced using the algorithm described above. The hope is that the window size will reach a steady state and at worst bounce between two window sizes.

Window Sizes Changes at 1 MSS

In chapter 6, we discuss a series of experiments which show that allowing the TCP window to shrink below 1 MSS causes extreme fluctuations in performance. This led us to modify our window sizing algorithm such that the window size would:

1. Grow and shrink by 1 MSS at a time
2. Never shrink below 1 MSS

The reasoning for this is shown in chapter 6.

5.2.2 Handling UDP

With UDP still being a large part of “average” network traffic, its bandwidth consumption and control needed to be dealt with. Unfortunately, by its very nature, it does not have any kind of traffic control mechanism that we could manipulate.

Our initial proposal called to use `ICMP_SOURCE_QUENCH` messages to control host to host communication. However, the host requirements RFC suggests that this mechanism not be used anymore. As a result, Linux does not acknowledge or send this type of ICMP message. Upon further investigation, we found that Sun Solaris does not use it either. Because of Sun’s dominance

in the industry, we felt that their decision not to support this message suggests that it would be useless to make it work under Linux.

The algorithm that we use to control UDP rates is no algorithm. We rely on the application itself to throttle its rate when it detects dropped packets. Inside the kernel, we run a state machine to determine when to drop packets and when to finally decide to forcefully close a connection. The state machine works as follows:

OKAY Deliver the packet. Check if we are over rate, if so, change to DROP.

DROP Update SNMP stats to say we dropped the packets and free the packet buffer. Check how long it has been since we started dropping packets. If the user specified drop time has elapsed, change into WARN. Reset the rate to zero.

WARN Deliver the packet. If the user specified UDP warn time has elapsed, check the rate. If the rate is too high, change state to KILL otherwise go to OKAY.

KILL Tell the stack that we received an ICMP_UNREACHABLE error and let it close the socket. Obviously the packet doesn't get delivered in the process...

5.3 The /proc Interface

During the initial development phases, most of the variables in the system were set at compile time by `#define` statements. In order to make performing experiments easier, an interface was established through the `/proc` file system.

`/proc` is a virtual file system that provides a gateway into the kernel [5]. Kernel threads can create directories and directory entries in the virtual file system. Hooks are then established so that when a file in `/proc` is read or written to a function is called inside the kernel to handle the request.

Several interfaces were made available through `/proc` for this thesis. They were:

`/proc/sys/net/ipv4/rate_acceptable_tcp` The intent of this entry was to make it possible for the user to specify in packets per second the threshold rate for a connection beyond which rate control mechanisms would have to be applied on it. This applied onto to TCP connections.

`/proc/sys/net/ipv4/rate_control_udp` This is the same as `rate_acceptable_tcp`, however this was for UDP only.

`/proc/sys/net/ipv4/rate_debug` A user configurable value to turn debugging code on or off inside the kernel. When turned on, the various components of the rate control system would send verbose messages to the `syslogd` daemon for logging.

`/proc/sys/net/ipv4/rate_microsec_per_chunk` This user configurable variable established how many microseconds a chunk would represent. Smaller values would give finer control at the expense of performance whereas larger values would give a more granular control which was fast responding. Because the history memory could not be resized during runtime, resetting this to a smaller value would also result in less history information being maintained.

`/proc/sys/net/ipv4/rate_status` An on/off switch for all of the rate control code. This was necessary during debugging since the process of downloading a new kernel would trigger not necessarily working rate control code.

`/proc/sys/net/ipv4/rate_udp_drop_time` A user configurable value for how long the UDP rate control state machine should stay in the DROP state. Time for this variable is measured in seconds.

`/proc/sys/net/ipv4/rate_udp_warn_time` A user configurable value for how long the UDP rate control state machine should wait in the WARN

state. Time for this variable is measured in seconds.

`/proc/net/tcp_rate` A table generated on demand of all of the currently opened TCP sockets, their source address/port, their destination address/port, their TCP state, and finally the last known rate of transfer.

`/proc/net/udp_rate` Same thing as `tcp_rate` except for UDP.

5.4 Interfacing with CBQ

Our ultimate goal with this project is to be able to establish rules on a per-host basis such that if a host needs a minimum amount of bandwidth for a particular stream, a combination of CBQ and our added control code could be used to guarantee the quality of service.

This integration was done in four steps: adding a `/proc` interface to CBQ so that reports on stream status could be shown in real time, associating socket numbers (TCP streams) to CBQ class identifiers, creating a `/proc` entry where users specify which stream they consider important, and modifying the TCP window size control code to be aware of whether it needs to slow a stream down or allow the stream to achieve best possible performance.

The first step of adding a `/proc` interface to CBQ was relatively simple.

The necessary code to generate a report from active variables in the kernel was added to `linux/net/sched/sch_cbq.c`. This code took CBQ class identification codes, broke them down into their components (major class ID, minor class ID) and then displayed them along with each class's unique class identification which is normally only used in kernel space. This report was available through `/proc/net/cbq`.

The second step of associating socket numbers to CBQ classes was also done in the `linux/net/sched/sch_cbq.c` file. This function accepted a socket number along with a class identifier as parameters and used them when calling the functions which determined the appropriate queue for outgoing packets based on route tables and user configured CBQ settings. This function returned the kernel unique class identifier.

The third step of creating another `/proc` entry for allowing users to specify data is straightforward.

The final step took the code we had written for controlling stream rates using TCP window sizing code and modified it again. The modifications made it such that if the user specified a CBQ stream as "important", the code would not artificially shrink the TCP window down thereby reducing throughput.

5.4.1 A Usage Example

An example of using these modifications is as follows: we create two CBQ classes, 1:1 and 1:2. Any packet destined for the video server's IP address is channeled through 1:2 which is set to use maximum available bandwidth. All other packets are channeled through 1:1 which is set to use a maximum of 500kbit/sec of bandwidth. Recall that these settings apply only to outgoing data.

The user issues the command to read `/proc/net/cbq` and sees that class 1:2 has a unique identifier of `0x3432CC2A`. He writes this unique identifier into our `/proc/sys/net/ipv4/cbq_class` file.

Now whenever data comes from the video server, the TCP window modifications do not artificially slow down packets, however, packets arriving from anywhere else do get slowed down so that packets from the video server enjoy more available bandwidth.

Chapter 6

Experimental Results

In this chapter we discuss what experiments were run using the modifications to Linux discussed in chapter 5.

6.1 Experimental Network Configuration

To test the modifications, we used two networks. The first was a small network on its own shared collision domain. The three hosts connected to the network were:

ford A 350Mhz Pentium II, running Linux 2.1.121 (no kernel modifications)

oid A 150Mhz Pentium Pro, running Linux 2.1.121 (with kernel modifications)

toybox A 180Mhz 604e based Macintosh, running MacOS 8.1

Both PC based Linux systems used 3Com 3C905 network interface cards and the Macintosh used its built in Ethernet interface.

All three hosts were connected to each other through a 10Mbit hub. No other hosts were connected during any of the experiments. Internet connectivity was provided through a PPP gateway on *ford*. The internet link was disabled during all experiments.

The second network was a small subset of hosts at the University of California, Riverside. The machines which we used were:

amx A K6-200 running Linux 2.1.119 (no kernel modifications)

nox A 90Mhz Pentium, running Linux 2.1.121 (with kernel modifications)

dioxin A 200Mhz Pentium running Linux 2.0.36 (no kernel modifications)

Amx and *dioxin* used 3Com 3C905 network interface cards and *nox* used a generic card based on the DEC Tulip DC21140 chip.

All three machines were connected to 100Mbit switches. *Nox* and *dioxin* were connected to the same switch. *Amx* was physically located 2 miles away and was 6 router hops from both *nox* and *dioxin*. Five of the hops were 100Mbit. The last hop was over a 1.541Mbps T1 link.

6.2 `ttcp` Over Ethernet

`ttcp` is a tool that measures the transfer rate (amongst other things) on a TCP or UDP connection. `ttcp` can be used as a user program as well as a server daemon. While in server mode, it listens to port 5037 waiting for a connection from another instance of `ttcp` working in client mode. Once connected, they transfer N buffers of K kilobyte length.

By default, `ttcp` reports statistics measured at both the sending and receiving hosts. These values may differ because of packet loss (so the receiver reports fewer bytes transferred than the sender) and/or differing views on the start or end time for the test. For each test `ttcp` reports:

- Total bytes transferred
- Total clock time to run. (the amount of time the user perceived the transfer to take)
- Number of CPU seconds consumed by the program
- Transfer rate in megabits of user data per “wall-clock” second
- CPU seconds consumed per Mbits/second of data transfer
- Total number of system calls
- Number of system calls per second

- Number of system calls per CPU second

In addition to the `ttcp` report, the modified kernel generates entries in its system logs showing the transfer rate in packets per second for each open socket. Along with the packets per second, the modified kernel also records changes in the TCP window size. *ford* initiated the connection. The kernel kept 3 seconds of history, expressed as a sequence of average values over consecutive "chunks", each representing 100 milliseconds.

6.2.1 Controlling `ttcp` to a Fixed Transfer Rate

Our goal in this experiment was to use the rate control modifications to the kernel to control the transfer rate to be 4Mbits/sec between *oid* and *ford*, down from an uncontrolled average rate of 6.5Mbits/sec. Looking at figure 6.1, we see that the rate control mechanism that we added to the kernel was able to force the average rate to the target value, which is shown as a dashed horizontal line. However, our algorithm was unable to reach a steady state in this case. On average, we saw about 3.5Mbits/sec. However without the even flow, this is not very useful.

In figure 6.2, we see how the window size changes during the same run as in figure 6.1. We observe that the window size oscillates in a similar pattern to figure 6.1. When the throughput was too high, the window size would drop

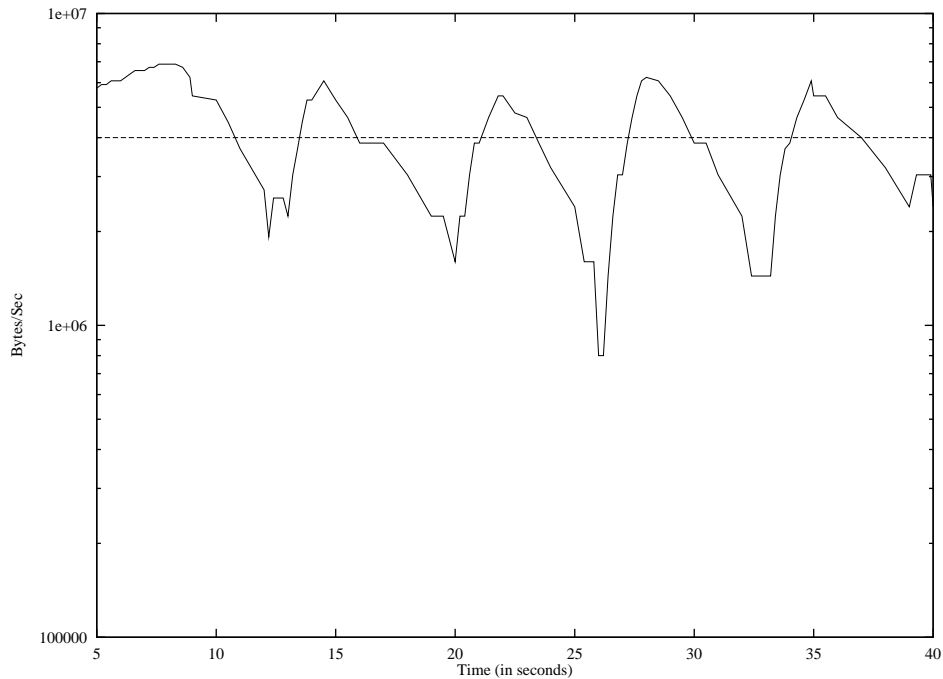


Figure 6.1: `ttcp` Experimental Result: Bytes Per Second

dramatically to constrain the transfer rate, and eventually the system would have to overcompensate to bring the rate down. But once the rate went below 3Mbits/sec, the window would expand and the rate would quickly jump again. This time the dashed horizontal line represents a window size of one MSS, the threshold below which we need to be concerned with Silly Window Syndrome.

Modifications to the control algorithm in which we attempted to control the rate fluctuations by tuning the amount of history data kept and the number of milliseconds represented by each chunk yielded almost identical

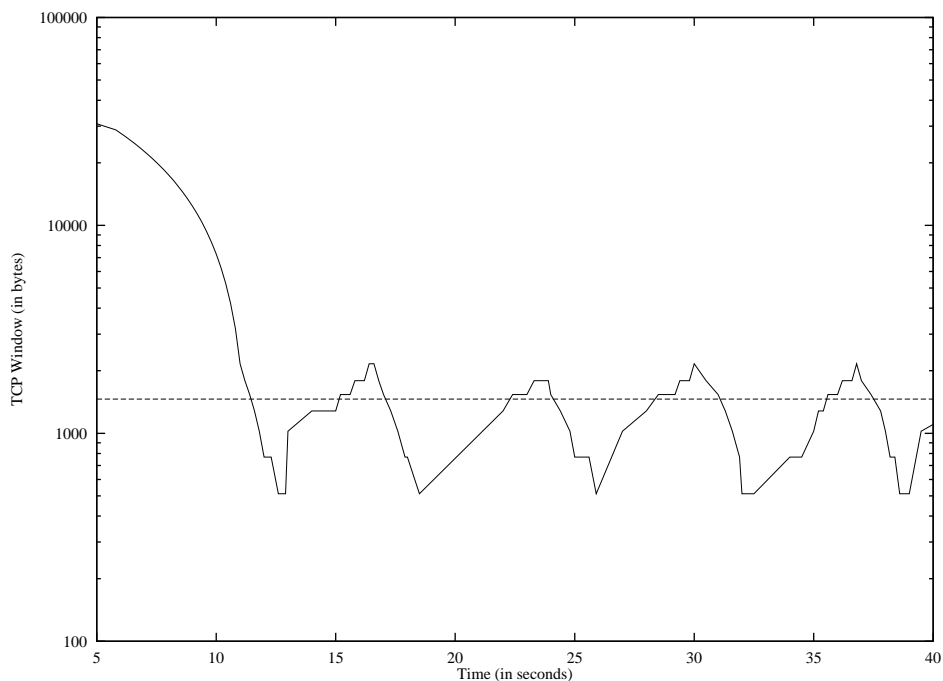


Figure 6.2: `ttcp` Experimental Result: TCP Window Size vs Time

results. Looking carefully at figures 6.1 and 6.2 we notice that the network throughput tended to come down more slowly than it went up, even though the window size would shrink more quickly than it grew. This suggested that there is a small range of window sizes around one MSS that would cause the network throughput to change drastically. Moreover, as shown in figure 6.3 which plots network throughput as a function of window size, the system exhibits considerable hysteresis whereby the same window size yields significantly higher throughput during the window shrinking phase than the window growth phase. In particular, notice how the curve forms a counterclockwise cyclic pattern around the intersection point between the

target throughput rate and a window size of about 1500 bytes. Obviously, the throughput for our system is very sensitive to window size and without greater understanding of the relationship between throughput and window size we cannot reach a specified steady throughput, and hence we cannot guarantee quality of service.

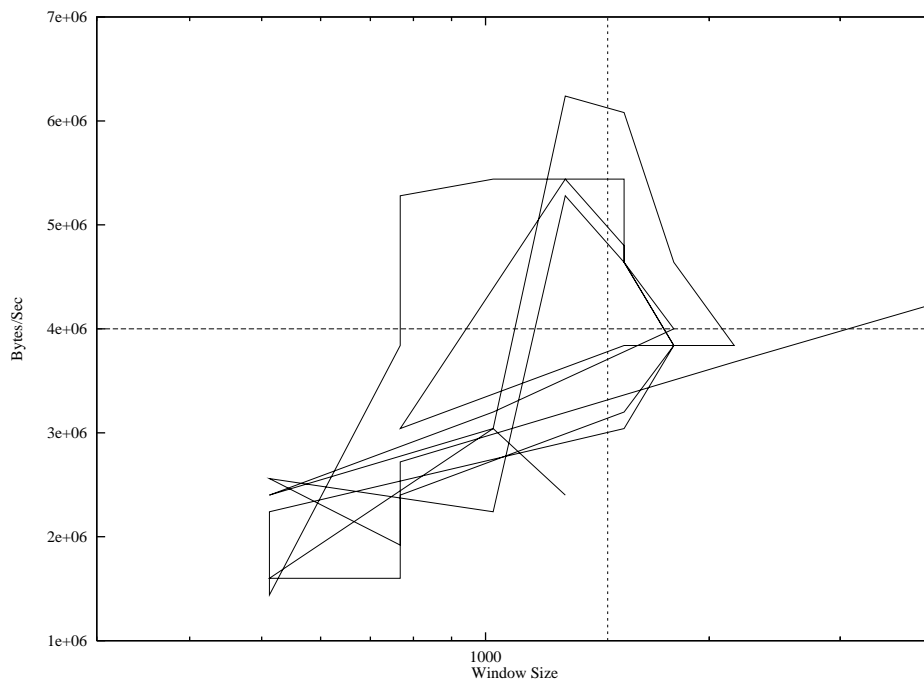


Figure 6.3: `ttcp` Window Size vs. Bytes/Sec

6.3 Sensitivity of Throughput to Window Size

The data from the `ttcp` experiment indicated that there was a drastic change in performance when the window size crossed some particular threshold value.

To find this point, we ran a second series of experiments in which we forced an application to use a user-specified constant window size and measured the resulting throughput rate in bytes per second. This was done by writing an application to download a file from a web server as we varied the window size.

We began by modifying the Linux 2.1.121 kernel so that it would have a `/proc` entry that would allow the window size be managed by a user rather than the kernel. The window size set by the user would be absolute, it would not grow or shrink.

With the kernel done, a simple web client was written to download files and measure the throughput in bytes per second. The host, *oid*, was configured to run the kernel and this client software.

On the other end of the wire was the host *ford*, which was configured to act as a normal web server. Ford ran a stock version of the 2.1.121 kernel and an unmodified version of the Apache 1.2.4 web server. The only file that was available through the web server was a 140M data file.

For the experiment, we tested window sizes from 32K bytes down to 1K in 1K increments. The results for each window size from 32K down to 2K were almost identical. To keep our graph less cluttered, figure 6.4 shows the performance for only window sizes 32K, 20K, 10K, 4K, 3K, 2K, and 1K.

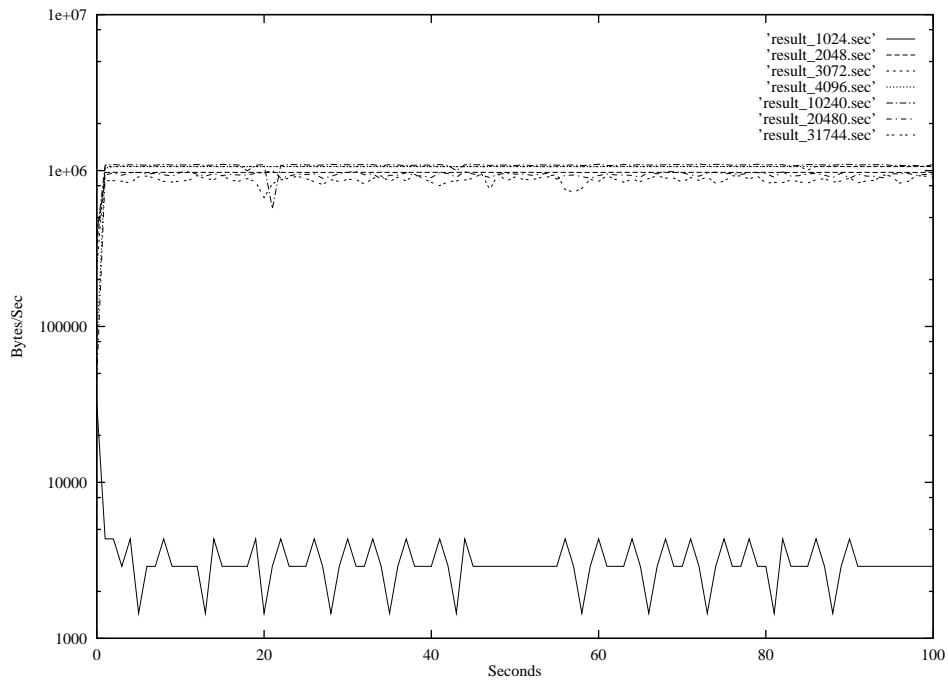


Figure 6.4: Bytes/Second for Varying Window Sizes (32K to 1K)

Figure 6.4 clearly shows that the window size of 1K resulted in a severe drop in performance. Because of the time it would have taken to transfer 140M at the rate shown for a 1K window size, the transfer was cut off at the 2000 second mark. In figure 6.4, we see a representative 100 second segment of the transmission.

To better examine what happened between the 2K and the 1K mark, we re-ran the experiment, this time going from 2K to 1K in 256 byte increments.

In figure 6.5 we see that the throughput of the transfer remained high up and through the window size reaching 1536 bytes. When the window size dropped to 1280 bytes, the performance fell to 2-4Kbytes per second.

When we observed the transfer using the `tcpdump` program, we found that when the window size dropped below 1448 bytes (the size of one maximum size Ethernet frame, minus the space needed for TCP/IP and Ethernet headers) the sender would send a 1448 byte block in two Ethernet packets. The first packet would be as large as the TCP window size, and the second large enough to fill the remainder of the Ethernet frame.

For example, if Alice (the server) were sending data to Bob (the client) and Bob were advertising a window size of 1400 bytes, a `tcpdump` of their transfer would effectively be:

Alice \Rightarrow Bob	DATA	sent 1400 bytes
Bob \Rightarrow Alice	ACK	Acknowledgement
Alice \Rightarrow Bob	DATA	sent 48 bytes
Bob \Rightarrow Alice	ACK	Acknowledgement
Alice \Rightarrow Bob	DATA	sent 1400 bytes
Bob \Rightarrow Alice	ACK	Acknowledgement
Alice \Rightarrow Bob	DATA	sent 48 bytes
Bob \Rightarrow Alice	ACK	Acknowledgement

This behavior is also known as the Silly Window Syndrome (SWS)[3].

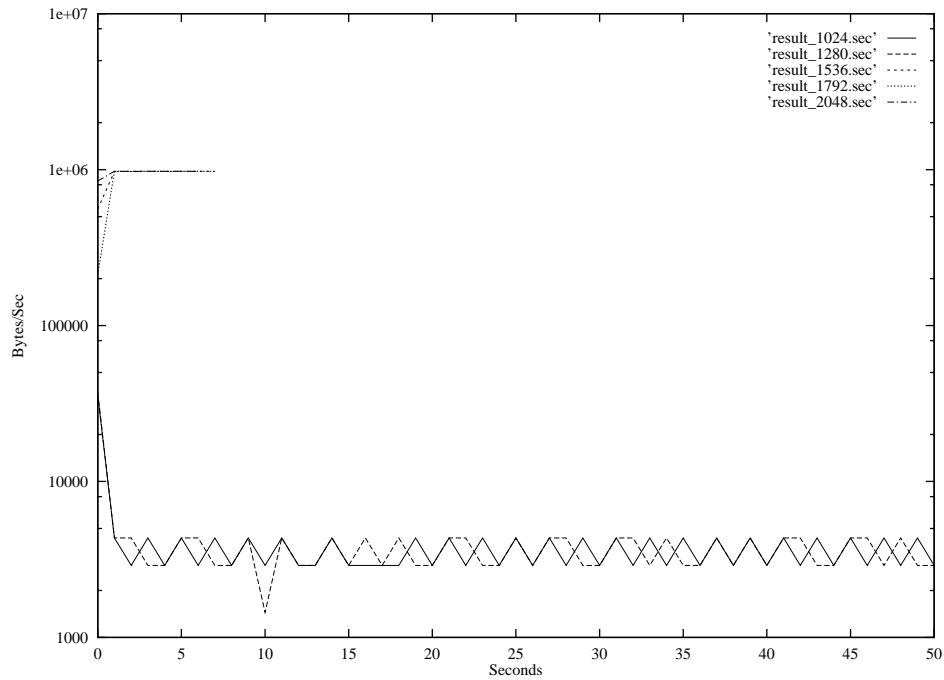


Figure 6.5: Bytes/Second for Varying Window Sizes (2K to 1K)

In chapter 4, we discussed SWS avoidance. We stated that TCP should avoid transmitting until one of three conditions are met:

1. a full MSS can be sent
2. we can send $\frac{1}{2}$ of the largest segment size ever advertised
3. we send everything we have while we are not expecting an ACK or Nagle's algorithm has been disabled.

In our case, the second rule, condition 2 would have been applied if we

did advertise a window larger than one MSS. However, since our receiver was not observing these rules, the sender was left confused and falling prey to SWS.

6.3.1 Findings

In short, this experiment showed us that TCP doesn't work well once the window size goes below the MSS. This explains the apparently bizarre behavior of the kernel modifications to control rate – it was specifying a window size below 1448 bytes and the rate was dropping from 900k/sec to 4k/sec. This kind of fluctuation of course makes it difficult to maintain an average!

This left us with two questions:

1. Since it is a TCP option to change the MSS which appears usable midstream, would changing the MSS along with the window size when the window size drops below the MSS make sense? Would it result in stopping SWS?
2. How many, if any, TCP/IP stacks would we break if we tried a stunt like that?

6.4 Evaluating the MSS Changeability

In the last section we observed that throughput dropped drastically when the TCP window size fell below the MSS of 1448 bytes. We identified the cause to be the Silly Window Syndrome (SWS), a case where TCP sends a small packet in conjunction with every large packet so that the two combined are worth one MSS of data. As a result, we waste a great deal of bandwidth and CPU time by transmitting the small packets.

This led us to the question of whether we can change the MSS midstream in TCP, not just in the SYN and SYN-ACK packets.

This experiment worked in two phases.

1. Add the necessary code to Linux 2.1.121 to advertise the MSS in each TCP header and verify that the system will behave the way we expect it to.
2. Tie in existing TCP window monitoring code inside the kernel so that should the TCP window fall below the current MSS, the MSS will shrink with it thus stopping the Silly Window Syndrome.

Once the code was written, we began a simple file transfer and observed each packet using `tcpdump`. We found that the packets were not being parsed correctly. Upon closer inspection and discussion regarding the matter with Alan Cox, one of the principle developers of the Linux networking code, we

found that it is not legal (or appropriate) to change the MSS once the stream is in motion. Other operating systems do not have to take any action if they do receive that option.

This answered the second question raised at the beginning of this section: TCP/IP stacks could, and rightfully so, not accept packets from my modified kernel if we changed the MSS midstream.

6.5 Connections with Differing MSS Values

In the last section, we learned that we cannot change an MSS midstream without causing TCP to misbehave. This lends itself to the question of TCP's behavior when run with differing start MSS values. The window size for each run was set to 1 MSS.

By default, the MSS of a TCP connection over Ethernet is 1460 bytes. We modified the kernel so that we could set the MSS for every new connection to be whatever value we chose. We ran a series of `ttcp` runs with differing start MSS values and observed the performance.

Our experiment was specifically setup as follows: *nox* was running the modified kernel. Dioxin connected to *nox* using `ttcp` and transferred 8Mbytes

over the link. Rate information was collected from both `ttcp`'s results as well as `nox`'s kernel logs.

We began with a series of runs to find the MSS at which performance is affected. We found that an MSS of 1000 bytes was the lowest setting before there was a visible change in performance. For this experiment, we consider the performance with an MSS of 1000 bytes to be 100% of the possible transmission speed.

6.5.1 Findings

We see in figure 6.6 that by reducing the MSS, we reduced the maximum throughput in an almost linear fashion.

To verify whether this behavior is reasonable, we recognize that TCP with a window size of one MSS behaves like a simple *stop-and-wait* data link protocol. In this case, we can easily compute the theoretical performance based on Ethernet's speed and our network's latency. This is calculated as follows:

1. Find the latency. This is done using the `ping` program which determines round trip time by sending an ICMP ECHO packet and timing how long it takes the other host to respond. In our test network, the

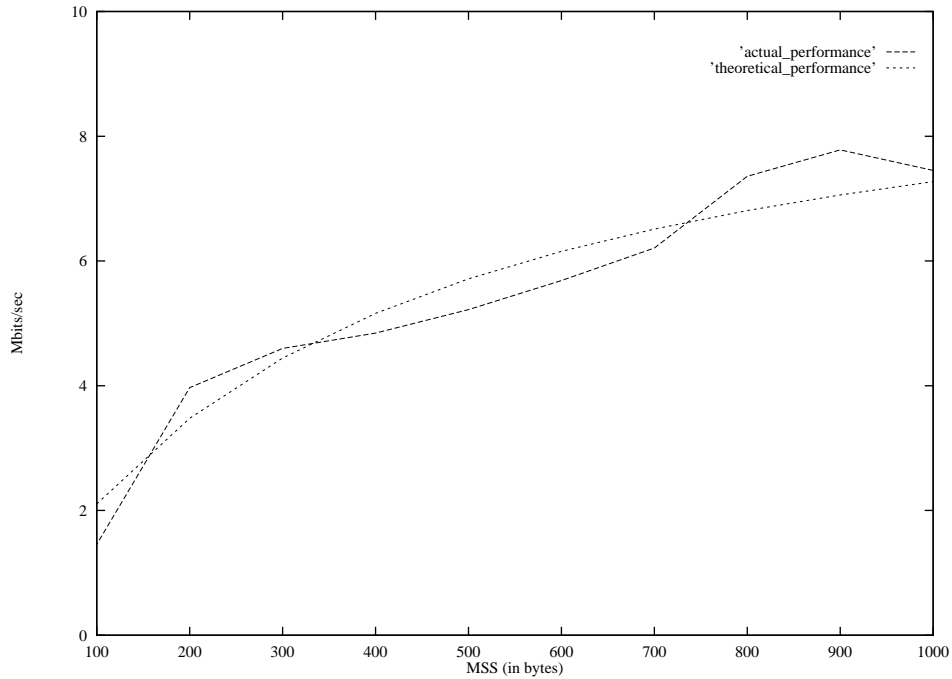


Figure 6.6: Bytes/Second for Varying MSS Sizes

average latency between *dioxin* and *nox* was found to be 0.3ms. This represents the round-trip delay between the end of a packet transmission by *dioxin* until it receives an ACK from *nox*.

2. We find the transit time for a 1 MSS packet. Using an MSS of 200 bytes for example, yields the equation

$$transmit_time = \frac{1,600bits(=200bytes)}{10,000,000bits/sec(10MbitEthernet)} = 0.2ms.$$

3. Taking the transmit time and dividing it by the total time (latency + transmit) yields the theoretical throughput for stop-and-wait. For our example, 3.47 Mbps.

We perform this computation for each MSS setting we ran the experiment on and plotted it against the actual data in figure 6.6. We see that the behavior of our algorithm matches the theory reasonably well for measured data.

Also from this graph, we see the potential to use MSS settings to better control rates over Ethernet assuming the user knows up front whether they want a connection to be rate controllable for QoS purposes, and if so, how much bandwidth it should take. However, as we can see from the graph, our ability to effectively control the bandwidth after the MSS goes below 200 bytes is limited.

6.6 Changing the Window Sizing Algorithm

As we discussed in chapter 5, we experimentally determined that we needed to revise our algorithm for sizing TCP windows. Based on the experiments done so far, we learned that allowing the window size to drop below 1 MSS causes severe performance issues and thus, our modified algorithm does not allow this to happen.

All of the remaining experiments in this chapter were done with the modified window sizing algorithm.

6.7 `ttcp` over a T1 Link

With what we learned in the previous experiments, we became curious about the behavior of our system over a slower link. To test this, we setup *nox*, a P90 running our version of the kernel at the College of Engineering, Center for Environmental Research and Technology (CE-CERT), and *amx*, a K6-200 running a non-modified version of the 2.1.119 kernel. Between *amx* and *nox* is 6 hops, one of which is a 1.541Mbps T1. All other connections are switched 100Mbit Ethernet.

We ran the following experiment four times: `ttcp` was run on *amx* and set to communicate with *nox*. *Amx* sent 8 megabytes and *nox* replied with 8 megabytes for a total of 16 megabytes transferred over the T1. The run was done 10 times, the first time with the system set to use 100% of the available bandwidth and each successive run taking 90%, 80%, etc.

6.7.1 Findings

We were pleased to see that we were able to control the transmission rate by varying the window size when the target rate was between 100% and 50% of the maximum line speed. More importantly, we did not fall prey to the Silly Window Syndrome.

Figure 6.7 shows a comparison between three `ttcp` streams and their second by second rates. The first stream, “run.100” represents the system when it was configured to allow the maximum possible traffic through (100%). “run.70” is at 70% and “run.50” is at 50% total capacity.

In addition, we computed the lowest theoretical rate at which we could control the stream using the algorithm discussed in section 6.5. In this case, our transmit time was 8ms for a 12,000 bit packet at 1,500 bit/msec. The RTT averaged 6ms. This line is labeled “theory.lowest”.

Figure 6.8 shows the average rate (in Mbps) of each run versus the percentage of the total bandwidth we set the system to use. On average, the system performed well with little variance between runs. Our baseline for maximum system throughput average 1.14Mbps and was done using `ttcp` between *amx* and *hono*, a 350Mhz Pentium-II with a 100Mbit 3C905 card at CE-CERT. The throughput went below about 0.6Mbps no matter how low

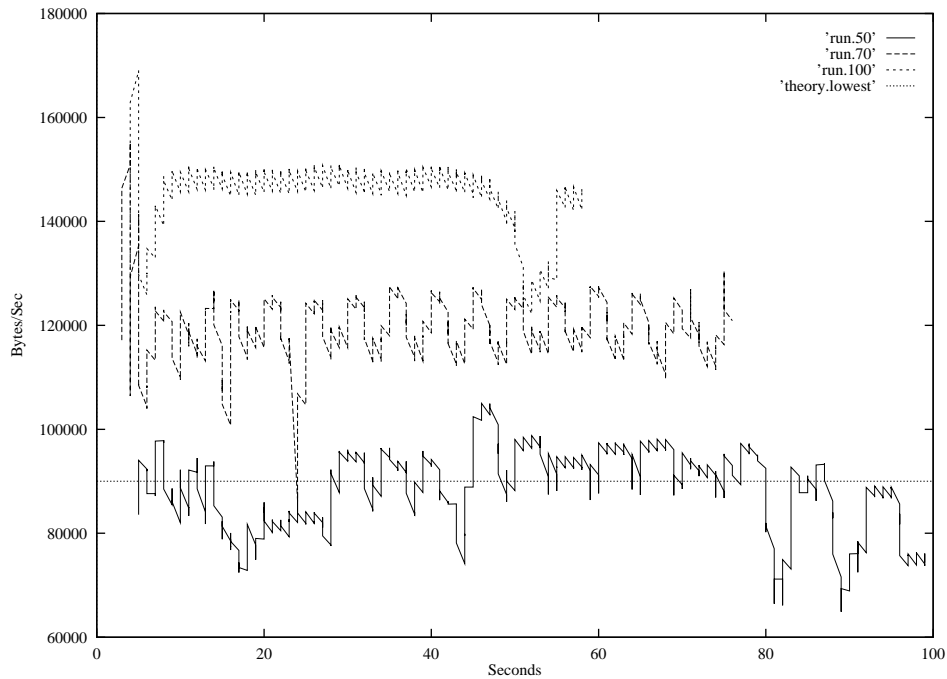


Figure 6.7: ttcp Rates Over a T1

we set the target percentage of bandwidth because the minimum window size was set to do one MSS.

6.8 ttcp Over A 38.8kbps Link

With the performance we saw over the T1, we decided to evaluate the performance of the algorithm at even lower speeds. To do this, a 38.8kbps PPP connection was established between *nox* and *dioxin* using a NULL serial cable. Route tables were setup so that any packets from *nox* to *dioxin* or vice versa would go over the PPP link.

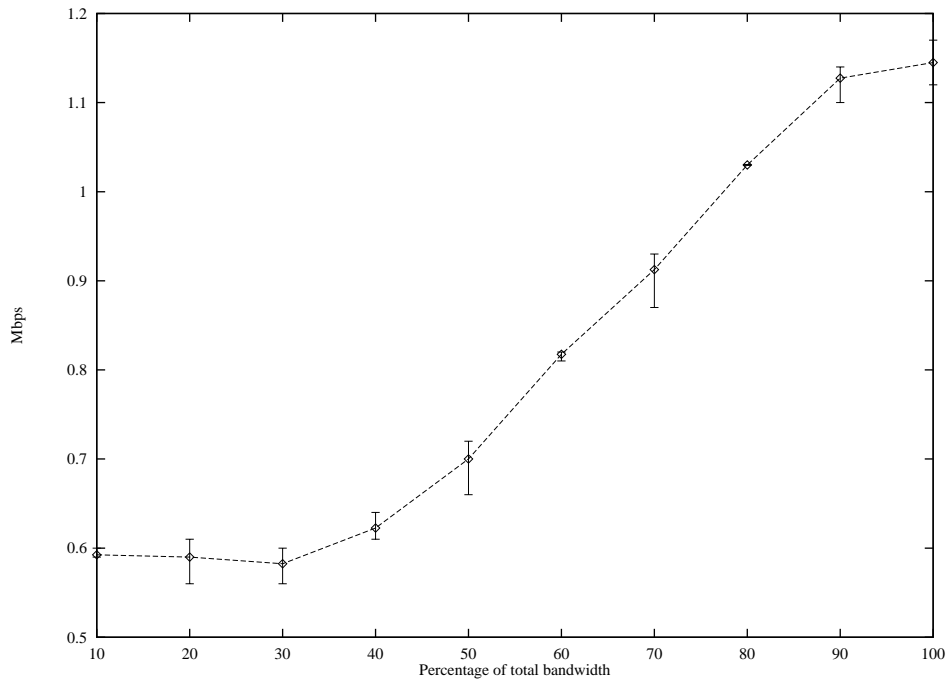


Figure 6.8: Average speed of 4 ttcp runs over a T1

The exact experiment was setup in a similar manner that the other `ttcp` experiments were setup. This time however, we specified the amount of data to be transferred from host to host to be 512Kbytes because of the substantially slower transfer rate.

Like the previous experiment, we started at 100% and worked our way down 10% at a time. We stopped at 30% because it had become clear the rate was not going to get any lower.

6.8.1 Findings

In general, we found that we were not able to accurately control bandwidth on this link. At best, the system would converge at a slightly less than optimal transfer rate, but would never go lower or higher. The problem here was that the transmission time for a 1500 byte frame over the serial line was so large compared to the propagation delay that stop-and-wait provided very high throughput.

What was very different about this experiment over the ones done with the T1 and Ethernet was the amount of time it took before a stream would reach convergence. As we can see in figure 6.9, it could take as long as 50 seconds before stabilizing at one rate. We believe this is because of the very high latency in the link (70ms versus 6ms for a T1 and 0.4ms for 10M Ethernet) and that it could take upwards of 2 seconds before a TCP ACK could be sent with a new window update.

As we can see in figure 6.9, streams which were set to utilize more of the link tended to spike quicker and then converge quicker. Because of the initial fast spike, it artificially makes it appear as if `ttcp` was able to transmit the same amount of data faster.

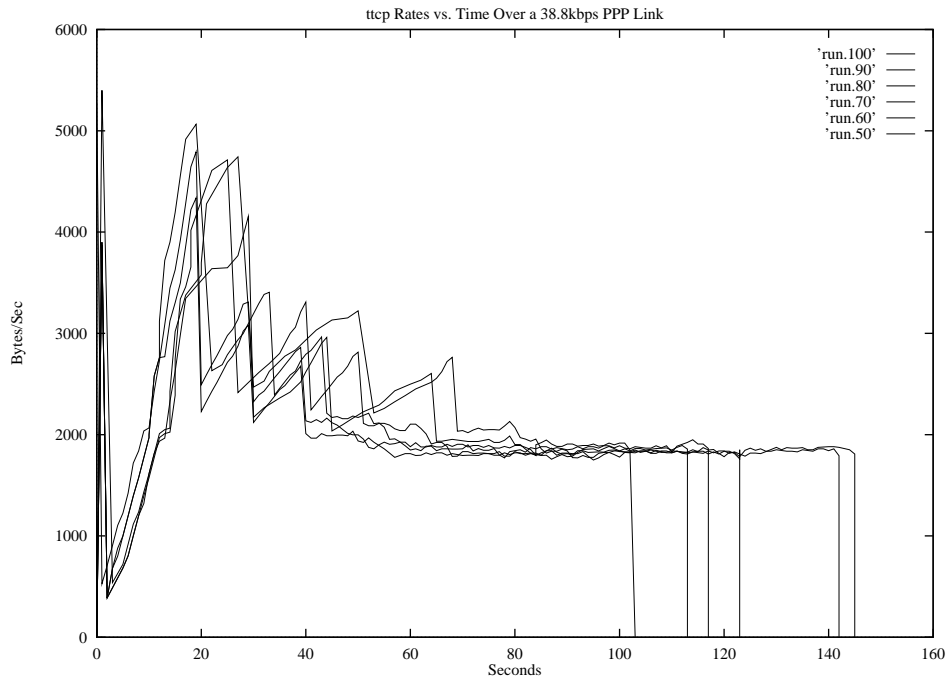


Figure 6.9: `ttcp` Rates versus Time in Seconds over a 38.8kbps PPP Link

6.9 Multiple Concurrent `ttcp`'s Over Ethernet

Since the previous two experiments showed that slower link speeds resulted in better results, we set out to find whether having to share a fast link (Ethernet) would eliminate the problem with the Silly Window Syndrome.

To test this, the host *dioxin* was setup to send multiple `ttcp` runs to *nox* at the same time over Ethernet. Runs were structured as follows:

- A single `ttcp` at 100% possible bandwidth was run.

- The amount of available bandwidth was shrunk by 20% for each run.
- This went from 100% down to 20% of the maximum possible rate.

This setup was then repeated with two `ttcp` runs, then three, and so on until we did a run with five concurrent `ttcp`'s.

6.9.1 Findings

Unfortunately, the Silly Window Syndrome remained with us even with five concurrent `ttcp` streams. We can see this with the second by second rate in one of the five concurrent runs with the requested bandwidth at 60% of total capacity in figure 6.10.

On average, the system did not help. `ttcp` reported 100% utilization of available bandwidth no matter what the setting. In the case of multiple streams, all of the streams combined together added up to 100% utilization. Figure 6.11 shows the average per-stream rate versus the number of streams running.

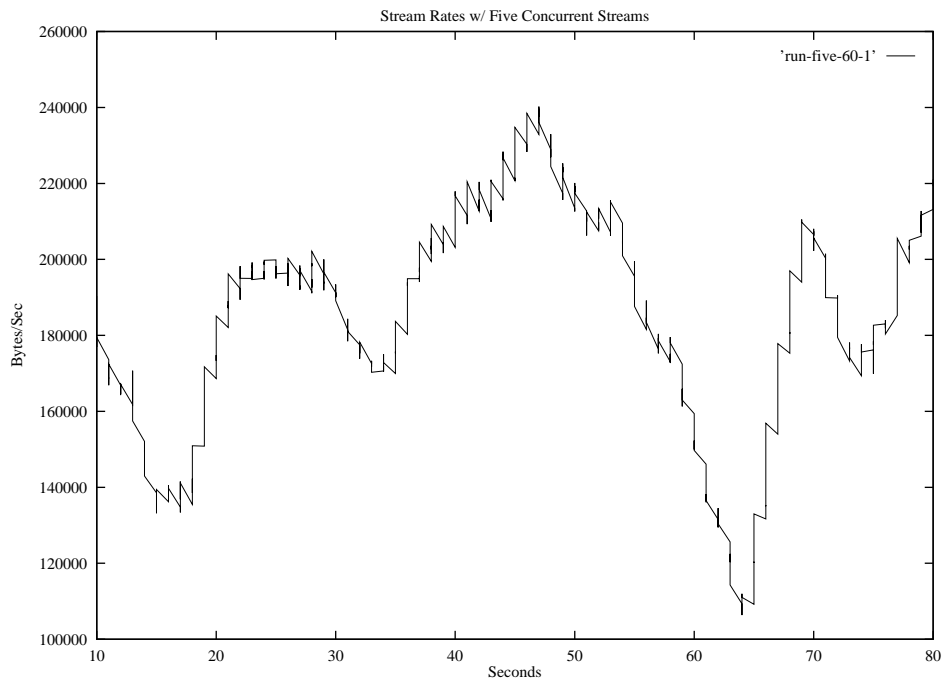


Figure 6.10: A single stream’s rate out of five concurrent streams at 60%

6.10 Multiple Concurrent `ttcp` over a T1 Link

In a configuration almost identical to the previous section, we tested the bandwidth utilization of multiple `ttcp` streams over a T1 link when the receiving host was configured to allow 100% of the available bandwidth, then 90%, 80%, and so on down to 10%.

This setup was done with one `ttcp`, then two, and so on up to five concurrent `ttcp`’s. We did four runs on each configuration.

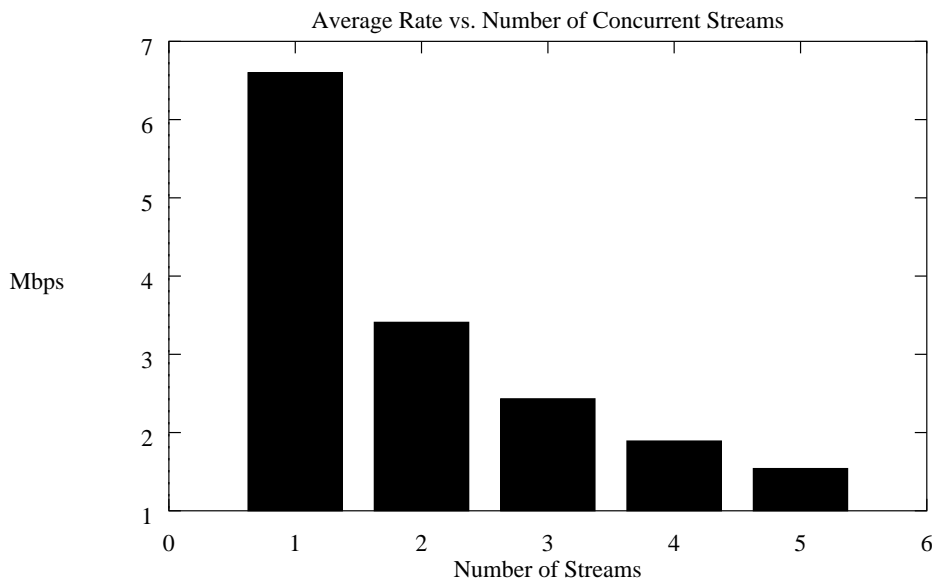


Figure 6.11: Average rate of a stream versus number of concurrent streams

6.10.1 Findings

Figure 6.12 shows what we expected given the experiments with multiple `ttcp`'s over Ethernet and a single `ttcp` over a T1. The bandwidth was shared equally amongst all of the streams, and when each individual stream required less than 50% of the available bandwidth, they were all shared equally regardless of total available bandwidth.

6.11 CBQ vs. Our Algorithm

In chapter 6, we discussed the nature and functionality of CBQ. With all of the functionality that the package offers, one might wonder why we have

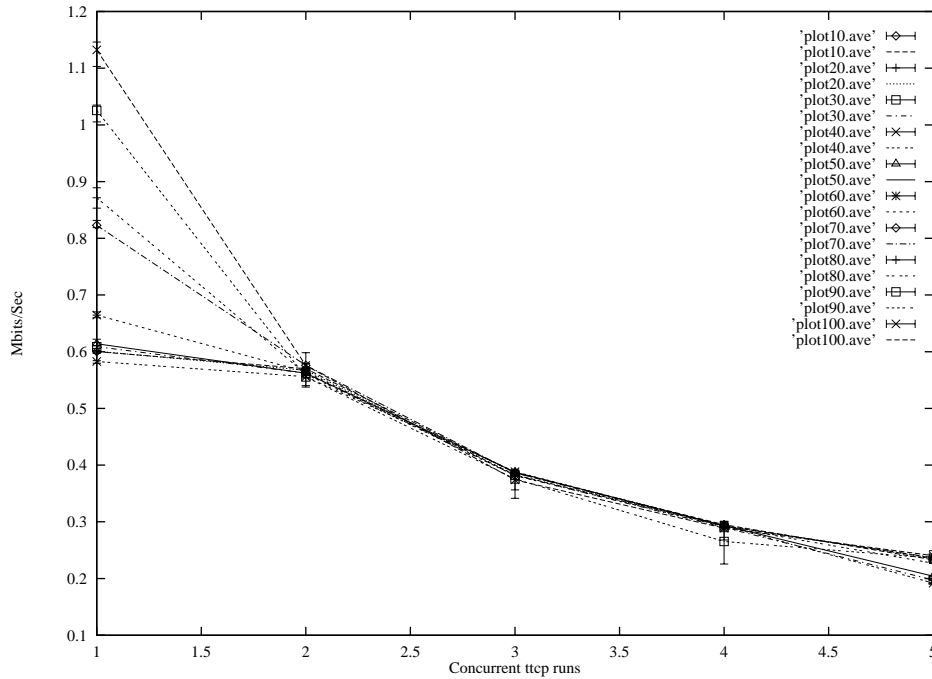


Figure 6.12: Multiple `ttcp` runs over a T1

gone through this thesis. The answer lies in the Linux source code.

All packets that are sent have the opportunity to go through the QoS mechanisms discussed in chapter 4. However, once a packet has arrived, it is guaranteed to be delivered (assuming there is enough memory). Thus, if Alice is sending a large stream of data to Bob, it is possible for Bob to become congested without an opportunity slow the traffic down.

One could argue that CBQ can be turned such that the return ACK packets sent to a host could be slowed down so that the round trip time increases

and the rate at which the data is being sent slows down. In this experiment we test this theory.

Our testing host, *nox*, is configured with all of our added algorithms disabled and CBQ enabled. Amx, which as we recall from previous experiments, is set to use `ttcp` to send data to *nox* over a T1 link.

We test 6 CBQ configurations. The first allows 100% throughput, the next allows 90%, then 80% and so forth down to 50%. For each configuration, we run `ttcp` four times.

6.11.1 Findings

As it is plainly visible in figure 6.13, using CBQ alone to slow down TCP acknowledgements is futile. The rate did not slow down at all and even when we asked to use only 50% of the available bandwidth, CBQ continued to utilize almost 100%.

From this, we believe that CBQ alone is not enough to control incoming TCP streams.

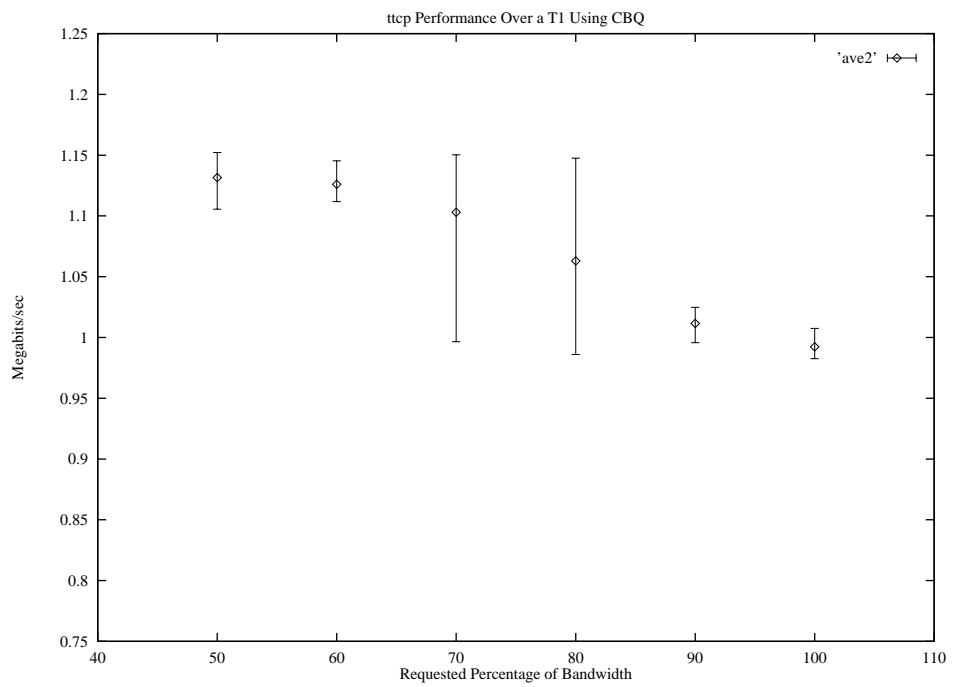


Figure 6.13: Performance when using CBQ to control incoming streams

Chapter 7

Conclusions

In this thesis we showed that it is possible to add *input* rate control to the Linux TCP protocol stack by manipulating the advertised window size as needed. In addition, we showed that such rate control can be done without the implementation complexity of additional timers or generating extra control packets as long as the target rate is above some system dependent lower bound. Whenever the advertised window dropped below one MSS, we observed a sudden, dramatic drop in performance that was caused by the Silly Window Syndrome (SWS) avoidance algorithm. Consequently, if we allowed our rate control algorithm to select window sizes smaller than one MSS, wild dynamic performance fluctuations would result. Forcing the minimum window size in our control policy to be one MSS solved the dynamic instability problem but meant that the minimum rate we can achieve with our control policy is simply the throughput for stop and wait which is easily calculated.

After trying numerous combinations of round-trip delay and bottleneck link speeds, we found that our approach was most effective for controlling bandwidth when the round-trip propagation delay is much greater than the packet transit time for the bottleneck link which allows for multiple packets to be in-flight. Satellite links, for example, would be very easy to control with our method. However, switched Ethernet and dial-up PPP would not.

Since changing the round-trip delay between a pair of communicating hosts is really not possible, we considered the opposite approach for improving controllability. We found that reducing the MSS did indeed allow us to reduce the rate for a flow. However, this change required knowing that the connection needed to be controllable for QoS purposes ahead of time.

Future Work

Based on what we learned from this research, we believe that the next step is to add timers so that delivery of window updates can be delayed in addition to window resizing.

Based on the importance of using a small MSS for controllability, future work should also include the construction of an API that allows you to specify that a high priority session is now in progress, so that any new low priority

connections would be forced to use a smaller MSS. To be effective, it may even be necessary to allow the user to kill and restart existing low priority sessions when a new high priority session is created.

Bibliography

- [1] Mike Borella. Ip grab 0.6 documentation.
- [2] R. Brandon. Rfc 1122. *Internet Engineering Task Force, Network Working Group*, 1989.
- [3] Clark. Rfc 813. 1982.
- [4] R. C. Crane and E. A. Taft. Practical considerations in ethernet local network design. In *Proceedings of the 13th Hawaii Intl. Conf. Syst. Sci.*, pages 166–174, Jan. 1980.
- [5] Steve Shah David Pitts, Billy Ball. *Red Hat Linux Unleashed 3rd Ed.* Sams, 1998.
- [6] R. Braden et. al. Ietf resource reservation protocol version 1 specification. December 1997.
- [7] Sally Floyd. Cbq/rsvp confusion note. 1996.
- [8] Sally Floyd. Ns2 simulator documentation. 1998.
- [9] Sally Floyd and Van Jacobson. Link sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995.
- [10] IETF DiffServ Working Group. A framework for differentiated services. <http://www.ietf.org/internet-drafts/draft-ietf-diffserv-framework-01.txt>, Oct. 1998.
- [11] R. Gusella. A measurement study of diskless workstation traffic on an ethernet. *IEEE Transactions on Communications*, 38(9):1557–1568, 1990.

- [12] Susan Hildreth. Balancing network resources. *Sun Expert Magazine*, 1998.
- [13] Red Hat Software Inc. About linux. http://www.redhat.com/linux_what.phtml, 1999.
- [14] R. Jain and W. R. Haw. Performance analysis and modeling of digital's networking architecture. *Digital Tech. Journal*, (2):25–34, Sept. 1986.
- [15] Paul Korzeniowski. Intranets still not ready for video. *Sun Expert Magazine*, 1998.
- [16] Alexey Kuznetsov. Linux cbq implementation: `/usr/src/linux/net/sched/`. 1998.
- [17] James A. Martin. Internet overload: Disaster in the making. *PC World*, Oct. 1996.
- [18] Mart Molle. A new binary logarithmic arbitration method for ethernet. *Technical Report CSRI-298*, April 1994.
- [19] Nagle. Rfc 896. 1984.
- [20] David Palmer-Stevens. *Cabletron Systems Guide to Local Area Networking*. Cabletron Systems, 1992.
- [21] Jim Pick. <http://www.linuxhq.com>. 1999.
- [22] Paul Russel and Michael Neuling. Ip chains documentation. `linux/net/ipv4/ipfw.c`, 1999.
- [23] J. F. Shoch and J. A. Hipp. Measured performance of an ethernet local network. *Communications of the ACM*, 23(12):711–721, Dec. 1980.
- [24] Stevens and Wright. *TCP/IP Illustrated Vol. 2*. Addison Wesley, 1995.
- [25] W. Richard Stevens. *UNIX Network Programming 1st Ed*. Addison Wesley, 1990.
- [26] W. Richard Stevens. *TCP/IP Illustrated Vol. 1*. Addison Wesley, 1994.
- [27] Wakeman and Ghosh. Implementing real time packet forwarding policies using streams. In *Usenix 1995 Technical Conference*, pages 71–82, 1995.